CRT-2000-44

Centre de recherche sur les transports (C.R.T.)
Centre for Research on Transportation

TEMPORAL SHORTEST PATHS: PARALLEL COMPUTING IMPLEMENTATIONS

by

Nicolas Tremblay
Michael Florian

December 2000

102-02893/

PUBLICATION
CRT-2000-44

*TEMPORAL SHORTEST PATHS: PARALLEL COMPUTING IMPLEMENTATIONS**

by

Nicolas Tremblay[1]
Michael Florian[1]

* This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

[1] Centre de recherche sur les transports, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Canada H3C 3J7

Dépôt légal – Bibliothèque nationale du Québec
Bibliothèque nationale du Canada, 2000

ABSTRACT

We explore two types of parallel computing implementations for three algorithms for computing temporal shortest paths on transportation networks. One implementation is done on a distributed network of SUN SPARC workstations using PVM and the other on a shared memory computing platform, a SUN SPARC server equipped with 8 processors, using threads. Computational results obtained by using three networks originating from practice are presented. The shared memory computing platform is preferred for this application.

Keywords: temporal shortest paths, parallel computing, PVM, multithreading.

RÉSUMÉ

Nous explorons deux types d'implantations parallèles de trois algorithmes de calcul des plus courts chemins temporels. Une implantation distribuée utilise un réseau de stations de travail SUN SPARC sous le protocole PVM et l'autre implantation est sur une plateforme à mémoire partagée, un serveur SUN SPARC à 8 processeurs en utilisant les "fils". Les résultats de calcul obtenus en utilisant trois réseaux provenant de trois villes au Canada sont présentés. La plateforme de calcul à mémoire partagée est préférée pour cette application.

Mots clefs: chemins plus courts temporels, calcul parallèle, PVM, multifils.

1. Introduction

The computation of temporal (or dynamic) shortest paths is of particular importance in the development of management and control measures of transportation networks. The recent interest in temporal network assignment methods such as DYNASMART (Mahmassani et al, [11]), DYNAMIT (Ben Akiva et al, [3], [1], [2]) and others, has motivated researchers in this field to develop algorithms which compute shortest paths efficiently. The principal difference between static shortest paths and temporal shortest paths is that the travel time on an arc is not constant, but depends on the time that the tail of the arc is reached. In this paper three recently developed temporal shortest paths algorithms are evaluated. The sequential versions of these algorithms are compared to two parallel computing implementations: one on a network of SUN SPARC workstations and one on a shared memory multi processor SUN SPARC server. The only previous contribution in this area are the parallel implementations of the label correcting method of Ziliaskopoulos and Mahmassani [13], [14] reported in [15]. They tested networks of up to 1,250 arcs on a CRAY Y-MP/8 and on a CRAY T3D with 32 processors using PVM. In our tests, this algorithm turned out to be the least efficient of the three algorithms that are considered here. Also, the speed-ups that we will report on the more accessible computing platforms which we used for our tests are based on computations carried out on three networks that originate from practice of a much larger size, varying from 2,500 to 17,000 arcs. These results are slightly better than those reported in [15].

The paper is structured as follows. In the next section the required notation is introduced and the three temporal shortest path algorithms are presented. Section three is dedicated to the description of the parallel implementations and the computational results are presented in section four. The final section is dedicated to some conclusions.

2. The Temporal Shortest Path Algorithms

Let $G = (N, A)$ denote a network composed of nodes $i \in N$ and arcs $a = (i, j) \in A$ where $i \in N, j \in N$. A travel time (or cost) function $d_{ij}(t)$ represents the time required at time t to reach node j from node i. $d_{ij}(t)$ is assumed to be a nonnegative function defined on discrete time intervals $S = \{t_0, t_1, t_2, ...t_{M-1}\}$, where t_0 is the smallest departure time for each node of the network and M is an integer sufficiently large so that $[t_0, t_{M-1}]$ is the period over which temporal shortest paths are relevant. $\pi_i(t)$ is defined to be the

shortest traversal time of the directed path which connects node i to the destination q at time t. $\Pi_i = [\pi_i(t_0), \pi_i(t_1), ...\pi_i(t_{M-1})]$ is the vector of travel times associated with node i for each time $t \in S$. Also, $b_i(t)$ is the successor node of i at time t on the temporal shortest paths to destination q and A_j^- is the set of arcs in the backward star of node j. The problem is to determine for each departure time t, the shortest paths from each node i to the destination q.

2.1 A label correcting algorithm

The algorithm proposed by Ziliaskopoulos and Mahmassani [13], [14] may be considered to be a temporal extension of the label correcting algorithms used to compute static shortest paths. The node labels are maintained in a list Q (which may be a *queue*, 2-*queue* or *dequeue* data structure).

This algorithm, which is referred to as TDLTP, performs the following steps:

TDLTP Algorithm

Step 1 (Initialization) :

$\pi_q = (0, 0, ..., 0)$, $\forall t \in S$;

$\pi_i = (\infty, \infty, ..., \infty)$, $\forall t \in S, i \neq q$;

$Q = \{q\}$.

Step 2 (Label Update) :

If $Q = \phi$, go to Step 4;

Otherwise, choose the first node j of the queue Q, $Q = Q - \{j\}$;

for each node $i | (i,j) \in A_j^-$ do

for each time period $t \in S$ do

if $\pi_i(t) > d_{ij}(t) + \pi_j(t + d_{ij}(t))$, then $\pi_i(t) = d_{ij}(t) + \pi_j(t + d_{ij}(t))$ and

$b_i(t) = j$

if at least one of the labels $\pi_i(t)$ was modified, then $Q = Q + \{i\}$.

Step 3 (Recursion) : Repeat Step 2.

Step 4 (End) : Stop.

At the end of the computations, the vector π_i associated to each node $i \in N$ contains the labels $\pi_i(t)$ for each time period $t \in S$. $S = \{t_0, t_0 + \delta, t_0 + 2\delta, ..., t_0 + (M-1)\delta\}$

is considered to be a set of discrete time intervals, where δ is a relatively small increment and M is sufficiently large such that $[t_0, t_0 + (M-1)\delta]$ covers the desired time period. The travel times $d_{ij}(t)$ are nonnegative real numbers defined on S; that is, $d_{ij}(\tilde{t}) = d_{ij}(t_0 + k\delta)$ for $\tilde{t}$ in the interval $t_0 + k\delta \leq \tilde{t} < t_0 + (k+1)\delta$. Also, $d_{ij}(t)$ for $t > t_o + (M-1)\delta$ is assumed to be constant and equal to $d_{ij}(t_0 + (M-1)\delta)$.

If the list Q is implemented as a simple queue, the complexity of the algorithm is of the order $O(n^3M^2)$. Initially, the queue Q contains only the destination q. During the first iteration at most $(n-1)$ nodes are inserted in Q. By executing $(n-1)$ repetitions of Step 2, one of these nodes will receive a permanent label. Each of these $(n-1)$ nodes may enter Q at most M times, since each node has M labels. Hence Step 2 will be executed at most $M(n-1)$ times. Since Step 2 requires $O(M(n-1))$ operations the algorithm has a complexity of $O(n^3M^2)$ in the worst case.

2.2 A label setting algorithm

The approach proposed by Chabini [6] does not require the use of any particular data structure. However, the travel times are assumed to be *positive integers* defined on $S = \{0, 1, 2, 3, ..., M-1\}$. This assumption is central to the development of this algorithm which will be referred to as DOT (for *Decreasing Order* of *Time*). It exploits implicitly the acyclic property of the time-space network obtained by a temporal expansion of the original network. Suppose that a static shortest path (SSP) algorithm is available. Then the DOT algorithm performs the following simple steps:

DOT Algorithm

Step 1 (Initialization) :

$\pi_i(t) = \infty, \ \forall (t < M-1, \ i \neq q)$;

$\pi_q(t) = 0, \ \forall t < M-1$;

$\pi_i(M-1) = SSP\,(d_{ij}(M-1), q)$ (where $d_{ij}(M-1)$ are the travel times at time $M-1$);

(Note that $\pi_i(t) = \pi_i(M-1), \forall t \geq M-1, i \neq q$).

Step 2 (Label Update) :

For $t = M - 2$ to 0 do

for all arcs $(i,j) \in A$ do

if $\pi_i(t) > d_{ij}(t) + \pi_j(t + d_{ij}(t))$, then $\pi_i(t) = d_{ij}(t) + \pi_j(t + d_{ij}(t))$ and $b_i(t) = j$.

As in the previous algorithm, when the algorithm terminates the vector π_i, associated to each node $i \in N$ contains the labels $\pi_i(t)$ for each time period $t \in S$.

The complexity of this algorithm is easily established. Step 1 requires nM operations to initialize $\pi_i(t)$, $\forall t < M - 1$, $i \neq q$ and requires the computation of static shortest paths for time period M. Step 2 requires $O(mM)$ operations. Hence the complexity of the algorithm is $O(SSP + nM + mM)$.

2.3 Another label setting algorithm

Pallottino and Scutellà [12] proposed, independently of Chabini, an algorithm which also exploits the acyclic property of the time-space network. This algorithm, denoted here CHRONOSPT, employs an inverse traversal of a *list of compartments* L in order to exploit this property. Each compartment corresponds to a time period t and contains the list of nodes to visit during that time period. The steps of the algorithm are as follows:

CHRONOSPT Algorithm

Step 1 (Initialization) :

$\pi_i(t) = \infty, \ \forall t \in S, i \neq q;$

$\pi_q(t) = 0, \ \forall t \in S;$

$L_t = \{q\}, \ \forall t \in S;$

$t_1 = M - 1.$

Step 2 (Label Update) :

While $L_{t_1} \neq \phi$ do

choose j from L_{t_1}; $L_{t_1} = L_{t_1} - \{j\}$;

for each $i | (i,j) \in A_j^-$ do

$t_2 = t_1 + d_{ij}(t_1)$;

if $t_2 > M - 1$ then go to 2

if $\pi_i(t_1) > d_{ij}(t_1) + \pi_j(t_2)$ then $\pi_i(t_1) = d_{ij}(t_1) + \pi_j(t_2)$ and $b_i(t) = j$

if $i \notin L_{t_1}$ then $L_{t_1} = L_{t_1} + \{i\}$.

Step 3 (Stopping Test) :

If $t_1 - 1 < 0$ go to Step 4

$t_1 = t_1 - 1$ and go to Step 2

Step 4 (End) : Stop.

Unlike the other two algorithms described above, this algorithm does not consider the arrival times which are larger than $M-1$, the last time period. As a consequence, certain labels $\pi_i(t)$ may stay at an infinite value if it is not possible to reach the destination before the last time period.

The complexity of this algorithm is $O(mM)$ in the worst case.

2.4 Implementation of the Algorithms and Evaluation Measures

The algorithms were coded in C++ by using a file format called NFF developed at the Center for Research on Transportation of the Université de Montréal (documentation available at http://www.crt.umontreal.ca/~lab_sit/DOC-NFF). This file format has been designed for the easy representation of transportation networks. The associated C++ library permits the efficient access to the forward and backward stars of nodes, which are operations frequently used in shortest path computations. An example of a simple network and the associated NFF files is given in the Appendix.

The data structure used in the implementation of the algorithm of Ziliaskopoulos and Mahmassani is a *dequeue*. In order to avoid the insertion of the same node in Q several times, a variable is associated with each node i which indicates if the node is contained or has been contained in Q. The information is kept in a table $EOQ[.]$. For a node i

$$EOQ\,[i] = \begin{cases} 1 & \text{if } i \in Q \\ -1 & \text{if } i \text{ has already been in } Q \\ 0 & \text{otherwise} \end{cases}$$

The implementation of the algorithm of Chabini requires a static shortest path algorithm which computes from the destination to all other nodes. A label correcting algorithm is used for that purpose. The node labels are kept in a table without any particular structure.

The implementation of the algorithm of Pallottino and Scutellà requires a list of compartments, as indicated in Figure 2.1.

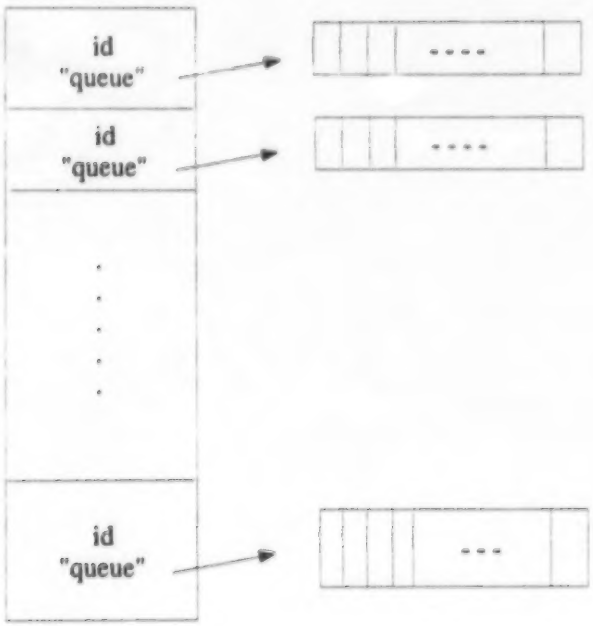


Figure 2.1 List of compartments

Each compartment contains an identifier (id) and a pointer (queue) to a *queue*. In order to add an element, it is necessary to specify the compartment identifier and the value that will be contained in the *queue* associated with the compartment.

Also, in order to avoid the insertion of the same nodes several times in the list of compartments, a variable $EOFQ\ [.][.]$ is associated with each node i and time period t which indicates if the node i is contained or has already been contained in Q. For a node i at time t the table is defined as follows:

$$EOFQ\ [t]\ [i] = \begin{cases} 1 & \text{if } i \in Q \text{ at time } t \\ -1 & \text{if } i \text{ has already been in } Q \text{ at time } t \\ 0 & \text{otherwise} \end{cases}$$

3. The Parallel Implementations

This section is dedicated to the parallel computing implementation of the three temporal shortest paths algorithms described in the previous section. Two approaches have been explored: one uses the PVM environment on a distributed network of workstations; the other is a *multithreaded* approach for a multi-processor shared memory machine.

3.1 The PVM implementation

The PVM (Parallel Virtual Machine) (Geist et al, [10]) is a message exchange protocol which permits distributed computation on a network of (possible) heterogeneous computing platforms. This development environment has two types of parallelizations: the *functional model* and the SIMD (Single-Instruction-Multiple-Data) model. In the sequel only the SIMD model is used.

The implementation which was developed uses the *master-slave* paradigm illustrated in Figure 3.1. It is composed of a *master* program and certain number of copies of a *slave* program. The task of the *master* program is to distribute the "work" to the slaves, to collect the results and to coordinate the execution of the "slave" machines. The tasks allocated to the *slave* program are relatively simple. It executes the "work" assigned by the *master* program and it communicates the results to the *master*. The message exchanges occur only between the *master* and the *slaves*.

The strategy used to parallelize the three temporal shortest path algorithms is straightforward. It consists in the division of the work by assigning one or more destinations to each *slave*. (This strategy was also used in [9]).

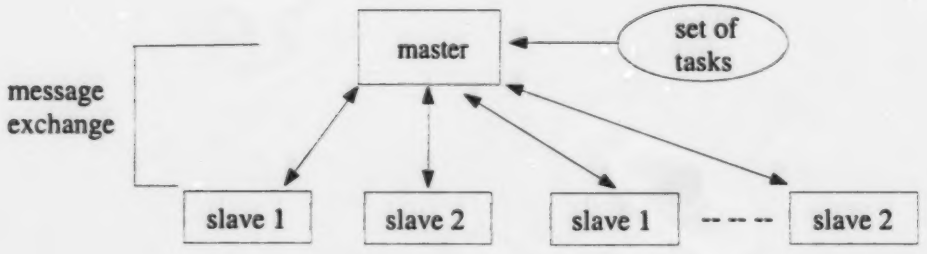


Figure 3.1 The master-slave scheme

The aim is to obtain the largest benefit of a modest number of processors (16 workstations, 8 processors, ...). The steps of the algorithm are not parallelized; rather, the global task is fragmented into subtasks with the aim of obtaining a high efficiency. Hence the computational grains are relatively coarse. The operations carried out by the *master* and *slaves* may be summarily described as follows:

master program:

[1] Reads the NFF file;

[2] Creates p copies of the *slave* program;

[3] Divides the set of destinations K into subsets k;

[4] Sends to each *slave* program a subset $k \in K$;

[5] While there is an active *slave* program,

5a. receives the results from the slave program.

5b. If all subsets k have not been allocated, sends a new subset $k \in K$ to the *slave* program i;
Otherwise, stop the execution of *slave* i.

slave program:

[1] Reads the NFF file;

[2] Creates the code for the computation of temporal shortest paths;

[3] While there is a subset of destinations to handle,

3a. receives from the *master* program a subset k of destinations or the instruction to stop execution;

3b. for each destination q, $q \in k$, computes the temporal shortest paths;

3c. sends the results to the *master* program.

The use of a message exchange environment implies that each slave must have a copy of the data necessary for the code that it is executing. As a consequence, the program *master* and *slave* must both read the NFF file of the network that they handle. This requires hence the duplication of information for each processor used.

The number of copies of the code *slave*, which are started by the code *master*, corresponds in general to the number of processors available for execution. In order to distribute the set of destinations $|K|$ among the available processors, the subdivision

$$\alpha \frac{|K|}{p}, \ \alpha \in [0, 1]$$

is used. This expression yields the number of subsets of destinations. The parameter α provides the possibility of the dynamic balancing of the total work load. When $\alpha = 1$ the number of subsets of destinations is equal to the number of processors. The number of destinations assigned to each processor is fixed and hence this is a static balancing of the computations. A value of $\alpha < 1$ results in a number $|k|$ of subsets of destinations

which is larger than the number of processors. In such a case it is possible to vary the load by assigning a subset $k \in K$ to the first free processor, which would favor the more efficient processor. This load balancing would also be beneficial if certain processors used are assigned to other tasks as well. The choice depends on the relative efficiency of the processors and their use for other purposes.

3.2 The *multithread* implementation

The use of *multithread* programming (Berg and Lewis, [4]) is particularly well suited for taking full advantage of a shared memory machine. On such a computing platform the communications correspond to read and write operations and are made via the shared memory. Unlike the PVM implementation, there is no data exchange since all data is *common*. However, the access to the data must be coordinated in order to avoid conflicts.

The multithread implementation uses as well the *master-slave* paradigm. Each *slave (thread)* is assigned the computation of shortest paths for a certain number of destinations. The distribution of tasks is achieved by a totally dynamic load balancing method. Each *thread* acquires, as soon as it finished its assigned computations, a new destination from the destinations that are yet to be handled.

Unlike the PVM implementation, where the *master* program and the *slave* program correspond to two distinct executable files, the use of *multithread* programming results in a single executable code. Nevertheless, it is possible to distinguish in this code the *master* part and the *slave* part. The operations executed for each part are the following:

master part:

[1] Reads the NFF file;

[2] Creates p *threads* and waits for the end of the computations.

slave part:

[1] While there is still a destination to handle,

1a. selects a destination not handled;

1b. computes the shortest-paths for the selected destination.

Multithread programming requires often a synchronization mechanism, when the access to data is shared by the *threads*. In this implementation the *mutual exclusion lock* provided by the *multithreads* library is used to allow one and only one thread to execute a distinct

shortest path code and then ensures that a destination is handled by only *one* thread at a time. As soon as a *thread* ends an execution for a destination, it attempts immediately to acquire another destination, if one is still available.

The *multithread* code developed may be executed on a machine of one or more processors. The number of *threads* which are initiated corresponds, by default, to the number of processors available. Hence the processors are used to best advantage. The execution of the *multithread* code on one machine with one processor behaves exactly as the corresponding sequential version of the algorithm, except for a slight overhead due to the synchronization mechanism.

3.3 Performance measures used for the parallel computing implementations

The performance of a parallel computing implementation of an algorithm may be evaluated by several measures (see [7], [8]). In this paper the main efficiency criterion is the *execution time*. The measure that are directly related are those of *speed-up*, *efficiency* and *relative burden*. In order to state these criteria let p denote the number of processors, n the problem size, $T_s(n)$ the execution time of the sequential implementation on one of the processors of the parallel computing platform and $T(n,p)$ the execution time of the parallel implementation on p processors in order to solve a problem instance of size n.

The (absolute) *speed-up* $a(n,p)$ is usually defined as

$$a(n,p) = \frac{T_s(n)}{T(n,p)} \tag{1}$$

and the *efficiency* as $\frac{a(n,p)}{p}$.

Ideally $a(n,p) = p$ but in practice $a(n,p) < p$. It is important to note that in the computation of the speed-up $T_s(n)$ is used and not $T(n,1)$, the execution time of the parallel code on 1 processor. Usually $T(n,1) \geq T_s(n)$ due to the additional overhead to the communication and synchronization mechanisms of the parallel code.

The *relative burden* is a new performance measure introduced by Chabini [5]. Ideally, the execution time of the parallel code should be $\frac{T_s(n)}{p}$. The *burden* of the parallel implementation consists of all the time overhead incurred by a parallel code such as: the communication delay between processors, the wait time of the processors, the execution time of the non parallel parts of the algorithm, the additional time due to the machine

instructions which are specific to the parallel code and the variation of computing time for a problem instance due to the different solution path of the parallel code, when compared with the sequential code.

The *relative burden* $b(n,p)$ is obtained by dividing the burden of the processors by $T_s(n)$. The resulting measure is

$$b(n,p) = \frac{T(n,p)}{T_s(n)} - \frac{1}{p}. \tag{2}$$

The use of the relative burden as a performance measure has the advantage that it may be used to estimate the best speed-up by using a larger number of processors p. Since,

$$a(n,p) = \frac{1}{1/p + b(n,p)} \tag{3}$$

then as $p \to +\infty$

$$a(n,p) \approx \frac{1}{b(n,p)} \tag{4}$$

4. Computational Results

The networks used for the computational experiments are the coded road networks of the cities of Winnipeg, Ottawa and Montreal. These networks are used for the purpose of transportation planning (with the EMME/2 software package, INRO, 1999). While the use of these networks is with static models, they were judged to be appropriate for experimentation with temporal shortest path methods studied in this paper. The following table gives the dimensions of these networks.

Attribute / City	Nodes	Directed Links	Destinations
Winnipeg	1,057	2,535	154
Ottawa	2,569	6,963	258
Montreal	6,906	17,157	699

The PVM distributed computing codes were tested as a network of 16 SUN SPARC Ultra 1/140 workstations operating at 140 Mhz, equipped with 64 Mb RAM and running

SOLARIS 2.5. The workstations were connected by a 100 Mbits/sec Ethernet network. The shared memory computing platform used was a SUN SPARC Server 1000 equipped with 8 processors, 256 Mb RAM operating at 140 Mhz and running SOLARIS 2.5 as well. All the codes used in this work were compiled by using the SUN C^{++} compiler.

4.1 The results obtained with the PVM implementation

In order to test the performance of the temporal shortest path codes on the distributed network of SUN SPARC Ultra 1 workstations, the Winnipeg and Ottawa networks were used with time intervals of 30, 60, 90 and 120 periods, as well as the Montreal network with time intervals corresponding to 30 and 60 periods (Intervals corresponding to 90 or 120 periods of time could not be run due to the large memory requirements).

Tables 4.1, 4.2 and 4.3 contain the execution times obtained for the different networks used (Winnipeg/30 time periods, etc...) and the number of workstations used. The execution time is measured in seconds and corresponds to the actual time required by the application. These results were obtained by fixing the parameter α to 1, which corresponds to a static load balancing. Since all the experiments were run under ideal conditions with no other tasks executing on the workstations, it was verified by numerous tests that under such conditions a dynamic load balancing ($\alpha < 1$) is not useful. Since a dynamic load balancing increases the number of required communications, it is useful only when the situation warrants it. Since all the workstations used were identical, it was not advantageous to use any other strategy but the static load balancing.

The preliminary tests carried out indicated that this parallel code is inefficient if each *slave* communicates its results to the *master*. The results for each *slave* are contained in three matrices of dimension $D_k \times 12n|M|$ bytes, where D_k is the number of destinations in the subset k. Even if only one of these matrices were transmitted, such as the arrival times at destinations, the speed-up obtained is about 2 on 15 processors. As a consequence, the results presented in Tables 4.1, 4.2 and 4.3 do not include any communication of results and serve to measure *only* the efficiency of the distributed computation of the temporal shortest paths.

Network /|M|	Number of processors														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Winnipeg/30	4.06	2.07	1.37	1.06	0.83	0.70	0.63	0.54	0.48	0.43	0.39	0.36	0.35	0.34	0.34
Winnipeg/60	8.84	4.48	3.02	2.29	1.81	1.52	1.29	1.17	1.04	0.94	0.82	0.78	0.66	0.65	0.58
Winnipeg/90	13.15	6.61	4.46	3.36	2.67	2.23	1.89	1.72	1.54	1.39	1.22	1.13	0.96	0.95	0.95
Winnipeg/120	16.15	8.20	5.55	4.16	3.31	2.78	2.35	2.13	1.92	1.72	1.52	1.41	1.29	1.20	1.18
Ottawa/30	20.95	10.50	7.00	5.49	4.37	3.70	3.19	2.75	2.49	2.24	2.08	1.89	1.72	1.64	1.48
Ottawa/60	44.87	22.57	15.08	11.36	9.18	7.54	6.49	5.79	5.09	4.58	4.21	3.86	3.50	3.32	3.15
Ottawa/90	69.04	31.38	21.21	17.32	14.00	11.63	9.94	8.99	7.82	6.99	6.46	5.93	5.37	5.13	4.86
Ottawa/120	73.24	36.72	24.49	18.92	14.90	12.29	10.61	9.81	8.32	7.46	7.05	6.32	5.84	5.62	5.61
Montréal/30	149.51	74.94	49.31	37.10	30.01	25.21	21.21	18.93	16.45	14.88	13.55	12.50	11.46	10.63	10.31
Montréal/60	290.14	147.34	97.43	74.08	58.75	49.93	42.77	37.31	33.02	29.83	26.19	24.10	23.29	21.53	20.29

Table 4.1 *PVM* Implementation: execution time of algorithm *DOT*

Network /|M|	Number of processors														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Winnipeg/30	2.96	1.49	0.99	0.73	0.60	0.49	0.43	0.38	0.34	0.30	0.28	0.27	0.25	0.24	0.23
Winnipeg/60	9.01	4.52	3.22	2.29	1.81	1.53	1.30	1.19	1.06	0.96	0.79	0.73	0.74	0.66	0.62
Winnipeg/90	13.62	6.84	4.64	3.50	2.79	2.36	2.03	1.82	1.65	1.48	1.37	1.27	1.20	1.14	1.09
Winnipeg/120	16.71	8.50	5.70	4.37	3.47	2.95	2.52	2.32	2.05	1.90	1.71	1.60	1.42	1.39	1.39
Ottawa/30	30.90	15.64	10.43	7.89	6.32	5.23	4.51	4.02	3.53	3.18	2.92	2.68	2.44	2.32	2.07
Ottawa/60	62.36	31.48	21.48	16.25	13.02	10.78	9.28	8.28	7.27	6.52	5.60	5.15	4.69	4.46	4.19
Ottawa/90	99.13	47.33	30.75	22.92	18.64	15.77	13.53	11.77	10.59	9.56	8.57	7.97	7.33	6.95	6.62
Ottawa/120	119.73	60.93	41.51	30.78	24.55	20.93	17.72	15.84	14.79	13.19	11.60	10.75	9.78	9.30	8.87
Montréal/30	226.20	113.95	75.52	57.38	45.76	37.85	32.57	28.68	25.44	22.80	21.13	19.30	17.64	16.76	14.78
Montréal/60	464.60	233.35	155.99	118.82	94.24	78.90	66.84	59.64	53.86	47.46	43.38	40.13	34.98	32.63	30.68

Table 4.2 *PVM* Implementation: execution time of algorithm *CHRONOSPT*

Network /|M|	Number of processors														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Winnipeg/30	14.33	7.27	4.82	3.62	2.91	2.46	2.09	1.87	1.69	1.51	1.36	1.26	1.12	1.07	1.02
Winnipeg/60	64.17	33.36	22.38	16.45	13.21	11.03	9.49	8.50	7.42	7.22	6.29	5.62	5.35	4.83	4.77
Winnipeg/90	110.26	55.91	37.03	28.59	22.56	18.86	16.08	14.08	12.48	12.29	10.45	9.57	9.09	8.08	7.95
Winnipeg/120	163.19	81.72	55.26	41.64	33.24	28.04	23.99	21.23	18.63	17.60	15.53	14.29	13.37	12.12	12.04
Ottawa/30	145.70	72.89	48.52	36.78	29.57	24.40	21.43	18.86	16.69	14.94	13.66	12.66	11.37	10.85	10.10
Ottawa/60	370.99	190.43	126.87	95.04	76.55	63.67	55.09	48.03	43.24	38.62	35.51	33.20	30.56	28.17	26.40
Ottawa/90	626.73	313.76	209.84	158.73	126.87	106.24	92.36	81.38	71.08	64.06	58.75	53.90	49.26	46.55	44.12
Ottawa/120	899.51	451.35	301.60	226.70	181.82	151.01	129.96	116.05	102.09	91.37	84.41	77.67	72.48	67.27	63.54
Montréal/30	1040.53	527.97	350.99	263.11	213.04	177.64	152.92	136.26	119.96	107.44	99.02	90.02	83.17	77.74	73.82
Montréal/60	2766.46	1363.65	911.71	685.97	566.31	472.40	403.41	350.91	314.38	280.44	253.81	244.51	225.41	198.30	201.55

Table 4.3 *PVM* Implementation: execution time of algorithm *TDLTP*

The results obtained permit the computation of the **speed-up** and the **relative burden**. Figure 4.2 to 4.11 indicate for each of the parallel implementation of the three algorithms (DOT, CHRONOSPT and TDLTP) the graphs of the speed-ups and the relative burden for each experiment (Winnipeg/30 time periods, Winnipeg/60 time periods, etc...). These graphs indicate that this parallel implementation is efficient. The speed-up curves are nearly linear for all the experiments and, in most cases, the speed-up obtained is about 13 for 15 processors. The relative burdens are less than 0.09 and the corresponding curves are decreasing as the number of processors increases.

This implies that the temporal shortest path computation represents a sufficient computational load to justify such a distributed application. In other words, the *granularity* of the subtasks obtained is sufficient to favor the use of multiple processors.

In the figures 4.2 to 4.11 it is interesting to note that it is easy to distinguish the curves of relative burden, while the speed-up curves are very similar. In each graph the curve of the relative burden associated with TDLTP is under that corresponding to CHRONOSPT which is under that obtained for DOT. This is due to the performance of the sequential versions of these algorithms. Other than this observation, the relative burden permits to estimate the best speed-up on an unlimited number of processors. These estimates depend on the stability of the speed-up and relative burden curves. The table (4.4) gives, for each considered algorithm the estimates obtained with (4). These are obtained by using the value of the relative burden with 15 processors.

Algorithm	W/30	W/60	W/90	W/120	O/30	O/60	O/90	O/120	M/30	M/60
DOT	50.0	166.7	142.9	142.9	100.0	58.8	47.6	90.9	100.0	100.0
CHRONOSPT	83.3	125.0	58.8	62.5	200.0	250.0	142.9	142.9	142.9	*1000.0*
TDLTP	166.7	111.1	166.7	142.9	250.0	250.0	200.0	200.0	200.0	142.0

Table 4.4 Estimation of the potential speed-up - PVM implementation

For instance, Figure 4.1 of the Appendix indicates, for the experiment Montreal/30 and for each of the considered algorithms an estimate of the speed-up curves that would be obtained if one had more than 15 processors available. This network was chosen since the speed-up and relative burden curves are relatively stable. The speed-ups are computed by using (4), where $b(n,p)$ is the relative burden obtained on 15 processors.

The curves obtained indicate that the PVM implementation could be efficient as well on a larger number of processors.

In summary, the use of PVM on a distributed network of workstations is justified if the computational load considered may be split into subtasks of sufficiently coarse granularity. Also, the communications between processors must imply a time which is relatively small compared to the total computation time. In the computation of temporal shortest paths, if one were to transmit the results from the *slaves* to the *master*, this parallelization scheme would be inefficient (see also [9]). Thus, the results obtained by the *slaves* would be kept localy and perhaps used later. This scheme would be advantageous if the *slaves* were to use their results for a second computing task and then communicate the final result to the *master*.

4.2 Computational results with the *multithread* implementation

The SUN SPARC Server 1000, described previously, was the shared memory computing platform used for these tests. The networks used were the Winnipeg, Ottawa and Montreal networks for 30, 60, 90 and 120 time periods. The results obtained are presented in Tables 4.5 to 4.7 for each network considered and the number of *threads* used. The execution time is measured in seconds and is the actual time required by the application.

Based on these results, the graphs of the corresponding *speed-ups* and *relative burden* are presented in Figures 4.13 to 4.24 of the Appendix. These graphs are a convincing evidence of the efficiency of this parallel implementation. An average speed-up of 7.5 is obtained when 8 *threads* are used. As it was done in the previous section, by using relation (3) and by assuming an unlimited number of processors the best obtainable speed-up is estimated. Table 4.8 presents, for each of the networks used, the estimates obtained by using the relative burden obtained on 8 processors. The curves in Figure 4.12 of the Appendix indicate clearly that the computing times are likely to be improved for each of the algorithms tested by using more processors. The curve associated with the DOT algorithm indicates a nearly linear improvement for a number of processors larger than 100. This behaviour is characteristic for a *multithread* implementation since this type of parallel computing is influenced but little by the reduction of the granularity of the subtasks.

In summary, the use of *multithread* programming, just like the PVM implementation, results in efficient versions of the temporal shortest path algorithms tested in this paper.

However, this strategy is more efficient than the PVM implementation since there is no need to communicate the results obtained by each thread.

Network/$\|M\|$	Number of *threads*							
	1	2	3	4	5	6	7	8
Winnipeg/30	9.02	4.90	3.44	2.31	2.00	1.82	1.36	1.31
Winnipeg/60	22.66	12.03	7.49	5.46	4.40	3.75	3.11	2.70
Winnipeg/90	39.13	18.90	12.39	9.11	7.48	6.13	5.24	4.57
Winnipeg/120	51.30	26.48	17.20	12.88	10.26	8.47	7.13	6.34
Ottawa/30	55.40	27.72	17.77	13.74	11.40	9.24	7.78	6.82
Ottawa/60	120.50	64.07	40.72	30.65	23.90	19.85	17.01	14.82
Ottawa/90	196.18	79.09	54.30	41.10	39.70	30.31	26.13	22.74
Ottawa/120	259.42	132.63	80.21	60.71	48.51	40.03	34.02	30.25
Montréal/30	487.07	239.70	156.03	120.26	94.82	78.20	67.10	58.76
Montréal/60	957.32	471.01	316.42	233.89	186.85	153.36	132.43	116.39
Montréal/90	1457.92	735.55	469.62	361.01	285.97	241.75	202.71	179.23
Montréal/120	1866.33	901.75	615.44	455.48	370.97	311.84	279.32	241.30

Table 4.5 *Multithread* implementation: execution times of algorithm *DOT*

Network/$\|M\|$	Number of *threads*							
	1	2	3	4	5	6	7	8
Winnipeg/30	9.07	4.90	3.43	2.31	2.23	1.69	1.51	1.33
Winnipeg/60	25.00	12.17	8.09	6.16	5.19	4.10	3.55	3.08
Winnipeg/90	40.91	20.48	12.82	9.75	7.86	6.90	6.03	5.21
Winnipeg/120	56.53	29.85	20.27	14.51	11.28	10.06	8.56	7.68
Ottawa/30	47.55	23.58	16.43	12.60	9.73	8.14	7.15	6.18
Ottawa/60	111.77	58.11	37.50	27.80	24.50	18.23	15.85	14.56
Ottawa/90	180.28	90.61	59.25	44.47	38.05	30.26	27.32	22.20
Ottawa/120	237.36	119.75	87.47	59.97	52.53	41.24	37.20	32.57
Montréal/30	383.43	193.25	129.32	97.79	80.95	78.84	58.06	50.52
Montréal/60	804.08	403.04	271.92	228.69	170.91	146.80	126.84	105.73
Montréal/90	1447.06	735.74	491.95	334.73	295.36	234.18	201.07	177.88
Montréal/120	2108.87	1157.77	698.82	512.19	410.18	359.31	330.32	323.91

Table 4.6 *Multithread* implementation: execution times of algorithm *CHRONOSPT*

Network/$\|M\|$	Number of *threads*							
	1	2	3	4	5	6	7	8
Winnipeg/30	42.48	26.19	20.61	11.61	9.65	10.20	6.90	5.98
Winnipeg/60	152.70	84.12	50.93	37.94	29.62	25.69	21.82	18.72
Winnipeg/90	284.17	141.36	93.70	70.94	56.49	46.57	40.05	35.49
Winnipeg/120	468.11	242.56	158.16	115.27	91.06	75.98	65.07	56.76
Ottawa/30	279.23	140.97	94.79	71.18	56.69	46.15	39.30	34.57
Ottawa/60	875.52	429.21	283.55	214.23	164.99	138.03	117.56	101.63
Ottawa/90	1528.86	766.68	510.08	379.60	300.69	248.61	220.84	187.90
Ottawa/120	2310.87	1142.58	760.91	571.56	441.80	366.84	313.55	273.27
Montréal/30	2554.18	1280.91	897.11	648.16	522.86	433.05	366.70	321.54
Montréal/60	7015.45	3512.11	2374.84	1756.78	1385.41	1153.34	996.37	900.35
Montréal/90	12649.70	6206.59	4214.83	3084.31	2459.92	2043.63	1727.24	1545.25
Montréal/120	19418.70	10105.30	6441.43	4841.89	3933.71	3287.49	2802.25	2414.32

Table 4.7 *Multithread* implementation: execution times of algorithm *TDLTP*

Algorithm	W/30	W/60	W/90	W/120	O/30	O/60
DOT	45.5	500.0	333.3	200.0	500.0	500.0
CHRONOSPT	41.5	200.0	83.3	71.4	142.9	83.3
TDLTP	55.6	500.0	333.3	*5000.0*	500.0	100.0

Algorithm	O/90	O/120	M/30	M/60	M/90	M/120
DOT	*5000.0*	200.0	500.0	142.9	500.0	142.9
CHRONOSPT	500.0	83.3	142.9	142.9	333.3	29.4
TDLTP	500.0	200.0	200.0	200.0	142.9	*5000.0*

Table 4.8 Estimation of the potential speed-up: *multithread* implementation

5. Conclusions

Based on the results obtained, it is possible to conclude that the *multithread programming* is the better option for the computation of temporal shortest paths. However, this conclusion is limited to the two computing platforms that were used for these tests.

The other interesting observation is that the best speed-ups were obtained for TDLTP, which is the least efficient algorithm when only one processor is used. This is a common

occurrence when evaluating the performance of parallel implementation. However, the total computation time of this algorithm was nearly one order of magnitude higher than the other two algorithms considered in spilt of the nearly linear speed-ups obtained.

A shared memory computing platform is most likely to yield an efficient result, since the overhead of communicating intermediate computational results would be minimal for these algorithms.

Parallel computing implementations which are based on topological subdivisions of the network were not explored in this work since early evaluations led to the conclusion that they are not likely to be more efficient than the decomposition by destination.

References

[1] M. Ben-Akiva, M. Bierlaire, J. Bottom, H.N. Koutsopoulos, R. Mishalani, Development of a route guidance generation system for real-time application, Proceedings of the 8th IFAC/IFIP/IFORS Symposium on Transportation Systems, Chania, Greece, 1997.

[2] M. Ben-Akiva, H.N. Koutsopoulos, R. Mishalani, DynaMIT: a simulation-based system for traffic prediction, paper presented at the DACCORD Short Term Forecasting Workshop, Delft, The Netherlands, (1998).

[3] M. Ben-Akiva, H.N. Koutsopoulos, A. Mukundan, A dynamic traffic model system for ATMS/ATIS operations, IVHS Journal 2(1) (1996) 1–19.

[4] D.J. Berg, B. Lewis, A guide to multithreaded programming, SunSoft Press, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[5] I. Chabini, Nouvelles méthodes séquentielles et parallèles pour l'optimisation de réseaux à coûts linéaires et convexes, publication No 986, Centre de recherche sur les transports, Université de Montréal, 1994.

[6] I. Chabini, A new algorithm for shortest paths in discrete dynamic network, 8th IFAC Symposium on Transportation Systems, Chania, Greece, 1997.

[7] I. Chabini, M. Florian, On performance measures of parallel algorithms, publication No CRT-95–14, Centre de recherche sur les transports, Université de Montréal, 1995.

[8] I. Chabini, B. Gendron, Parallel performance measures revisited, High Performance Computing Symposium'95, pp. 381–392, Canada's Ninth Annual High Performance Computing Symposium and Exhibition, Montréal, Canada, July 10–12, 1995.

[9] M. Florian, I. Chabini, E. Le Saux, Parallel and distributed computation of shortest routes and network equilibrium models, 8th IFAC Symposium on Transportation Systems, Chania, Greece, 1997.

[10] A. Geist et al., Parallel virtual machine, a users' guide and tutorial for networked parallel computing, the MIT Press, Cambridge, Massachusetts, 1994.

[11] H.S. Mahmassani, S. Peeta, T.Y. Hu, A.K. Ziliaskopoulos, Dynamic traffic assignment and simulation procedures for ATIS/ATMS applications, Technical Report DTFH61–90–R-00074–FINAL, Center for Transportation Research, The University of Texas at Austin, 1993.

[12] S. Pallottino, M.G. Scutellà, Shortest path algorithms in transportation models: classical and innovative aspects, Equilibrium and Advanced Transportation Modelling, P. Marcotte and Sang Nguyen (eds), Kluwer Academic Publishers, (1998) 245–281.

[13] A.K. Ziliaskopoulos, H.S. Mahmassani, Design and implementation of a shortest path algorithm with time-dependent arc costs, Proc. of 5th Advanced Technology Conference, Washington, D.C., (1992) 1072–1093.

[14] A.K. Ziliaskopoulos, H.S. Mahmassani, A time-dependent shortest-path algorithm for real-time intelligent vehicle/highway systems applications, Transportation Research Record 1408 (1993) 94–100.

[15] A.K. Ziliaskopoulos, H.S. Mahmassani, D. Kotzinos, Design and implementation of parallel time-dependent least time algorithms for intelligent transportation systems applications, Transportation Research C, Vol. 5 (1997) 95–107.

Appendix

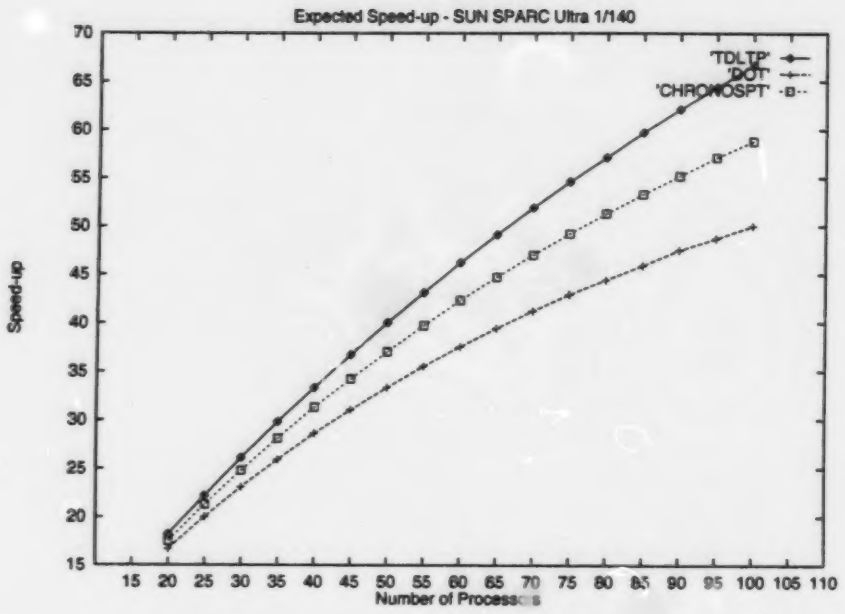
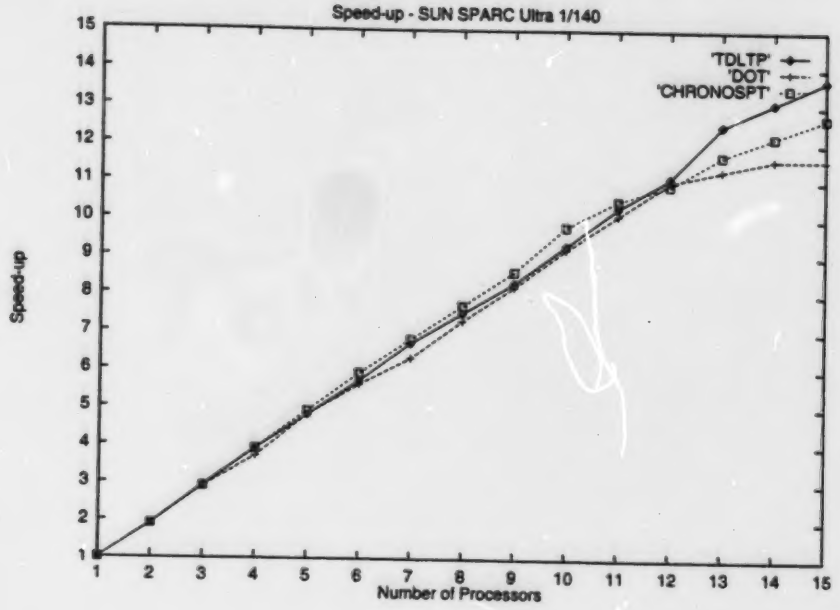


Figure 4.1 Estimated acceleration for more than 15 processors



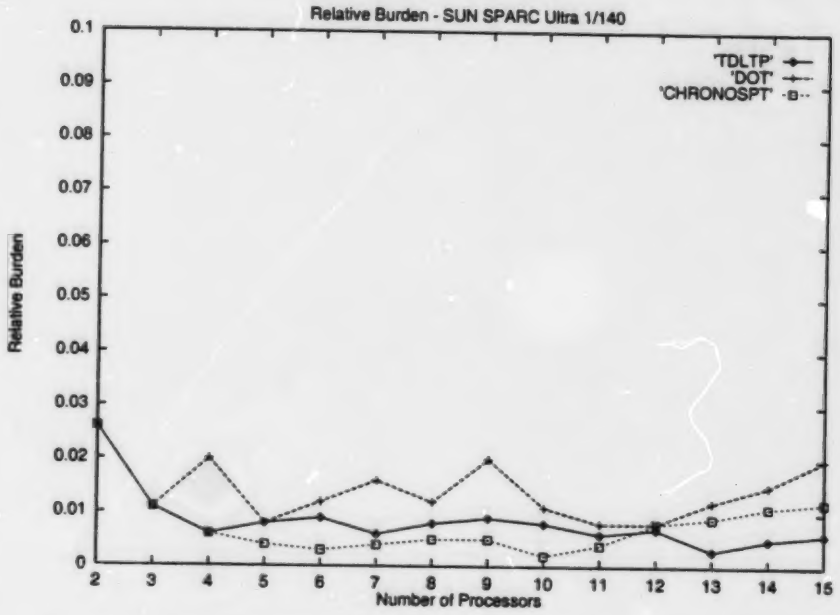


Figure 4.2 Speed-up and relative burden - Winnipeg, 30 time periods

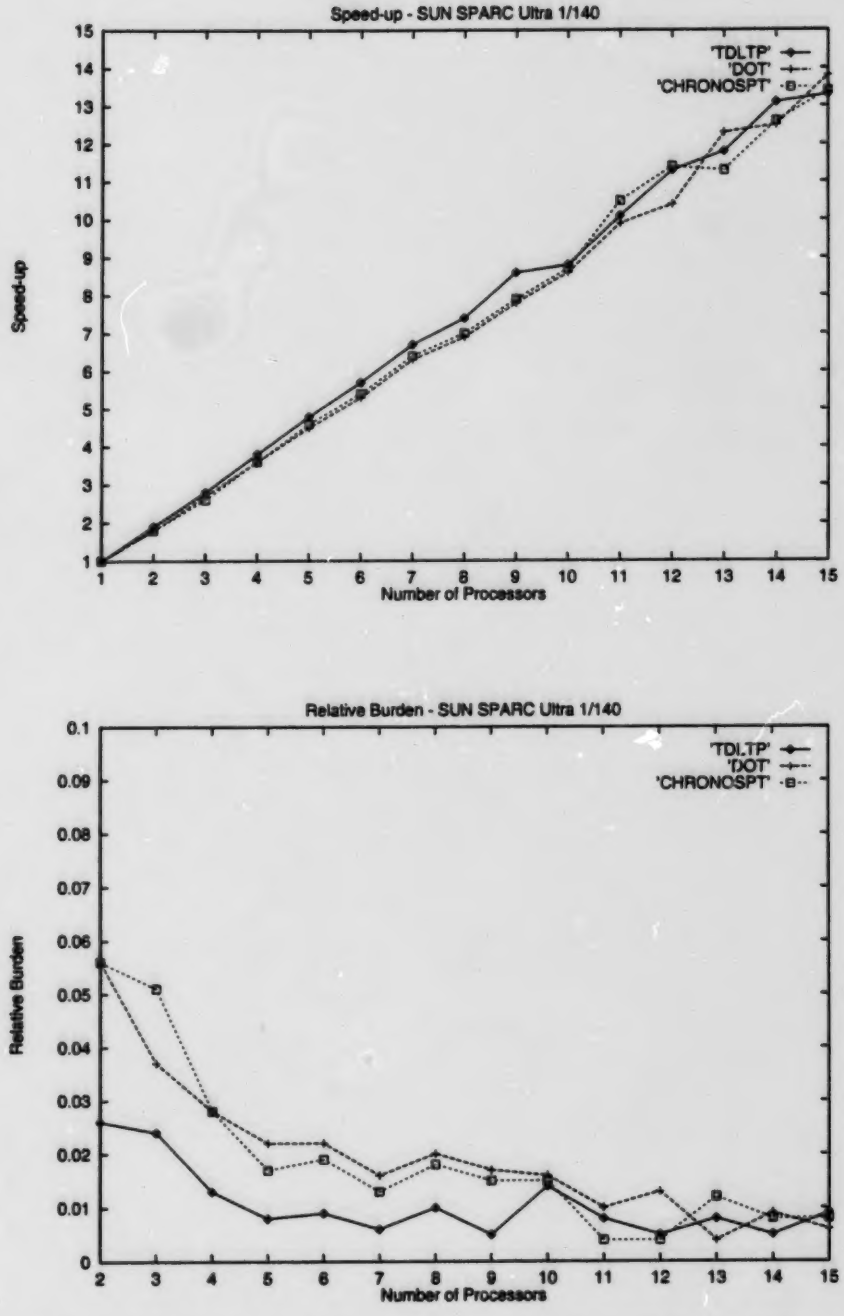
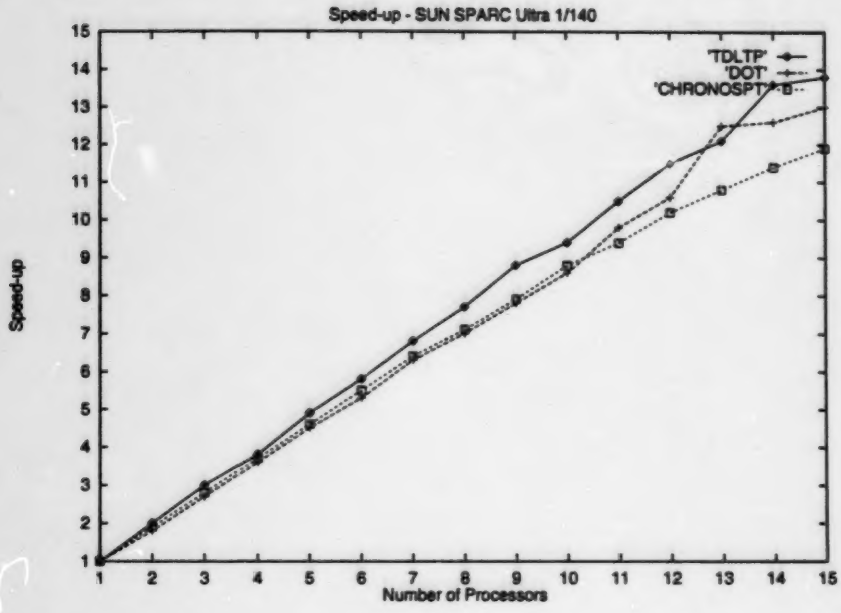


Figure 4.3 Speed-up and relative burden - Winnipeg, 60 time periods



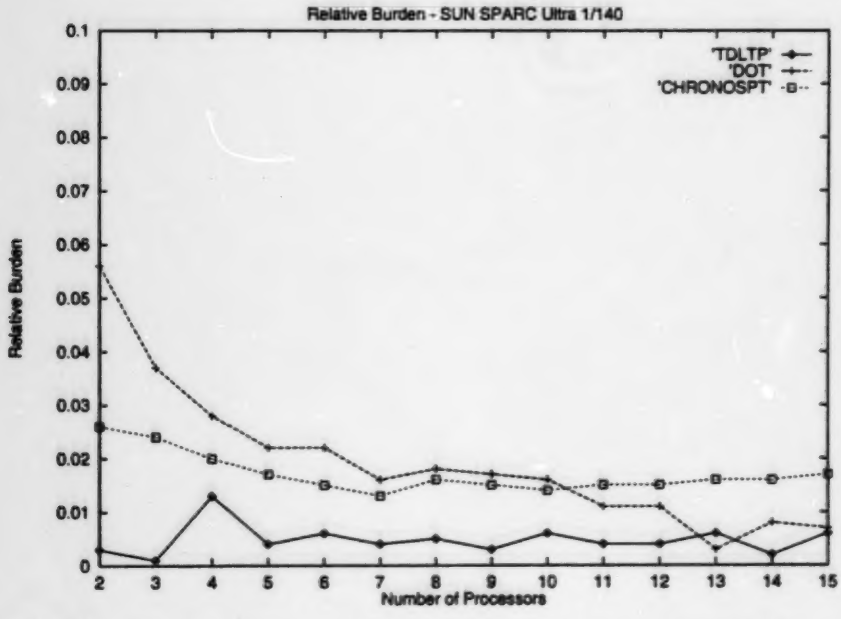
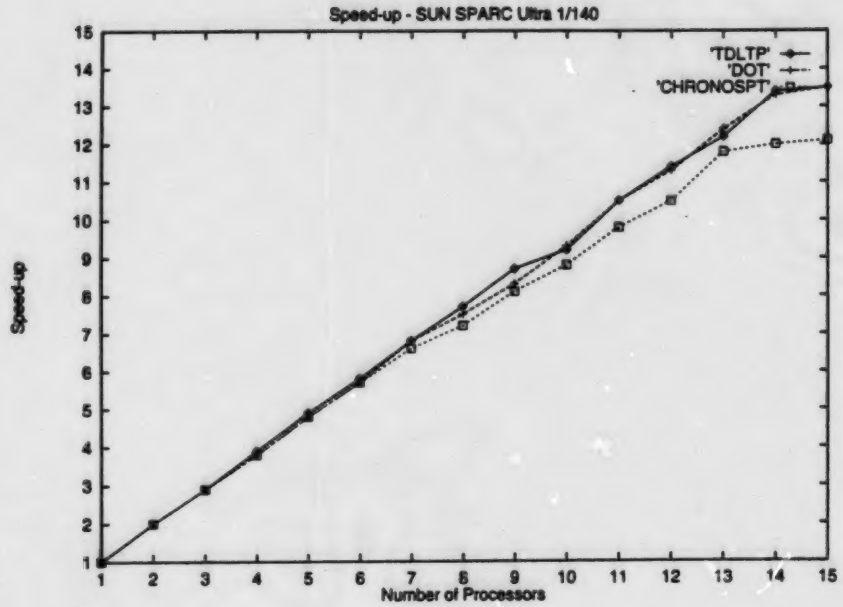


Figure 4.4 Speed-up and relative burden - Winnipeg, 90 time periods



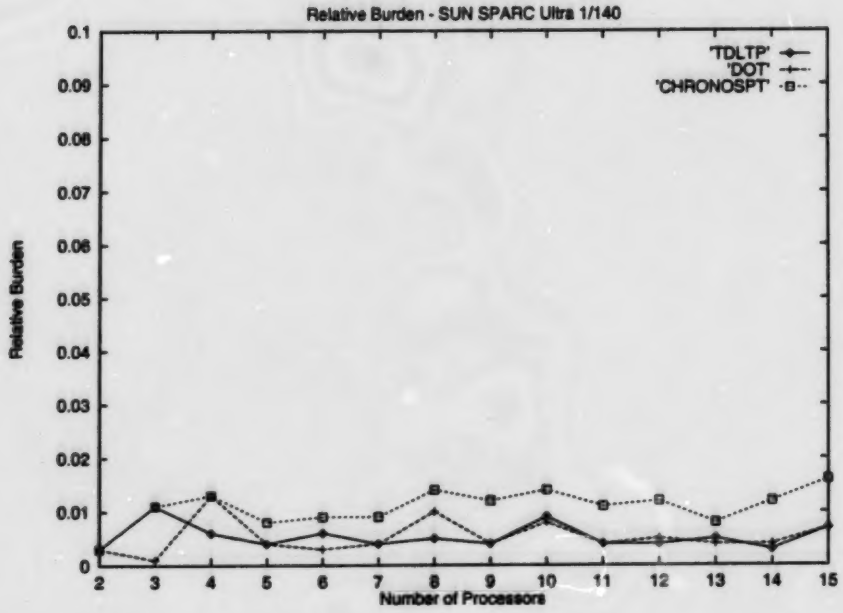
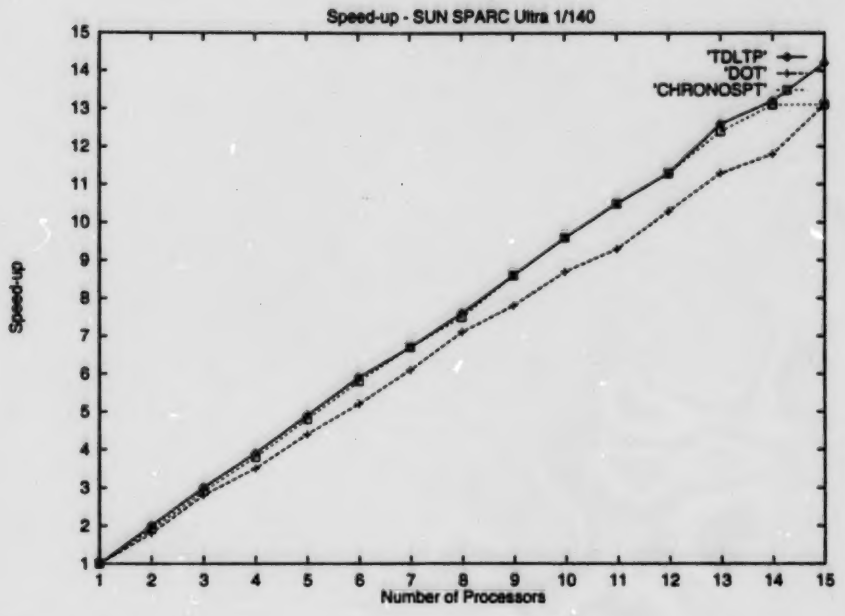


Figure 4.5 Speed-up and relative burden - Winnipeg, 120 time periods



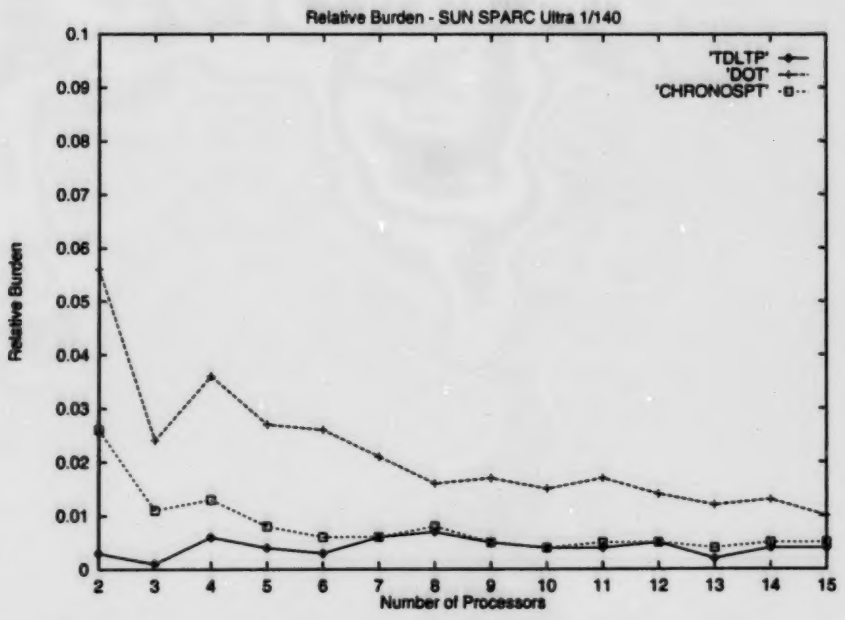
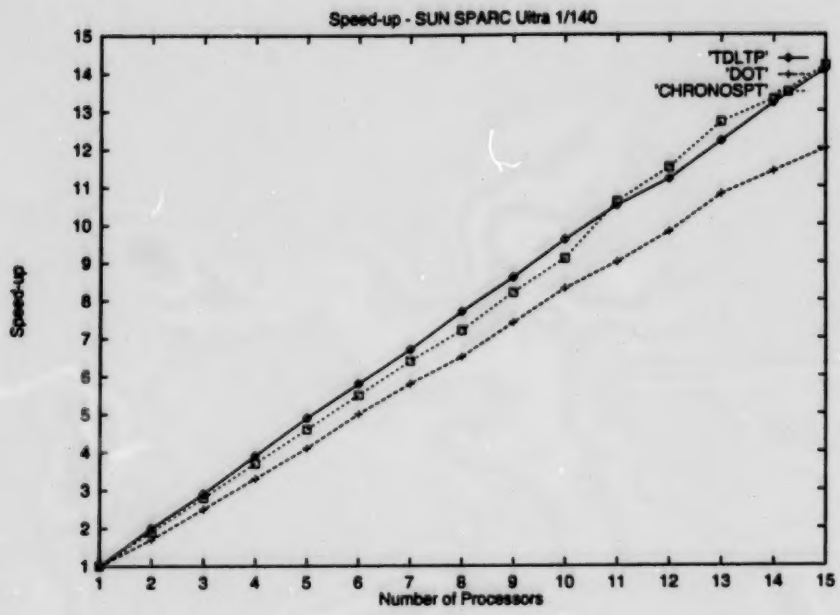


Figure 4.6 Speed-up and relative burden - Ottawa, 30 time periods



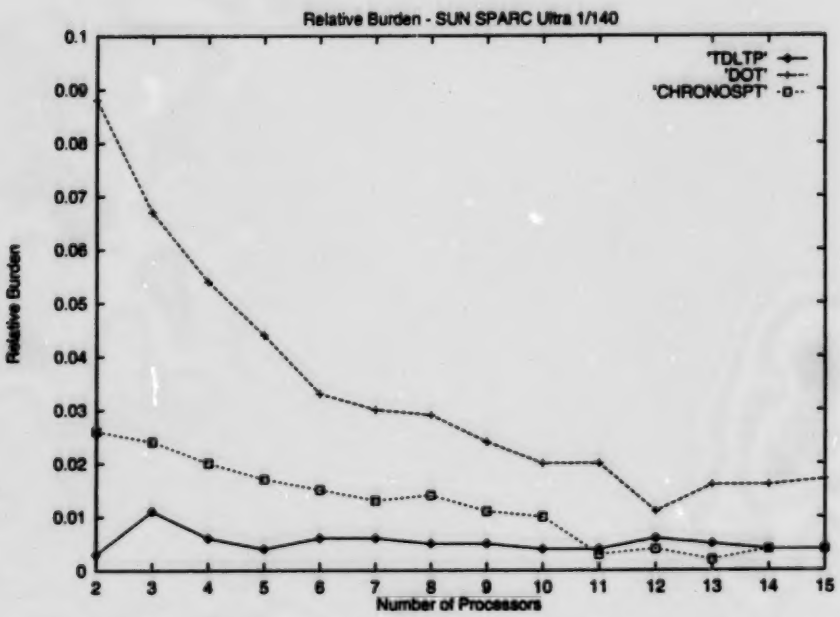


Figure 4.7 Speed-up and relative burden - Ottawa, 60 time periods

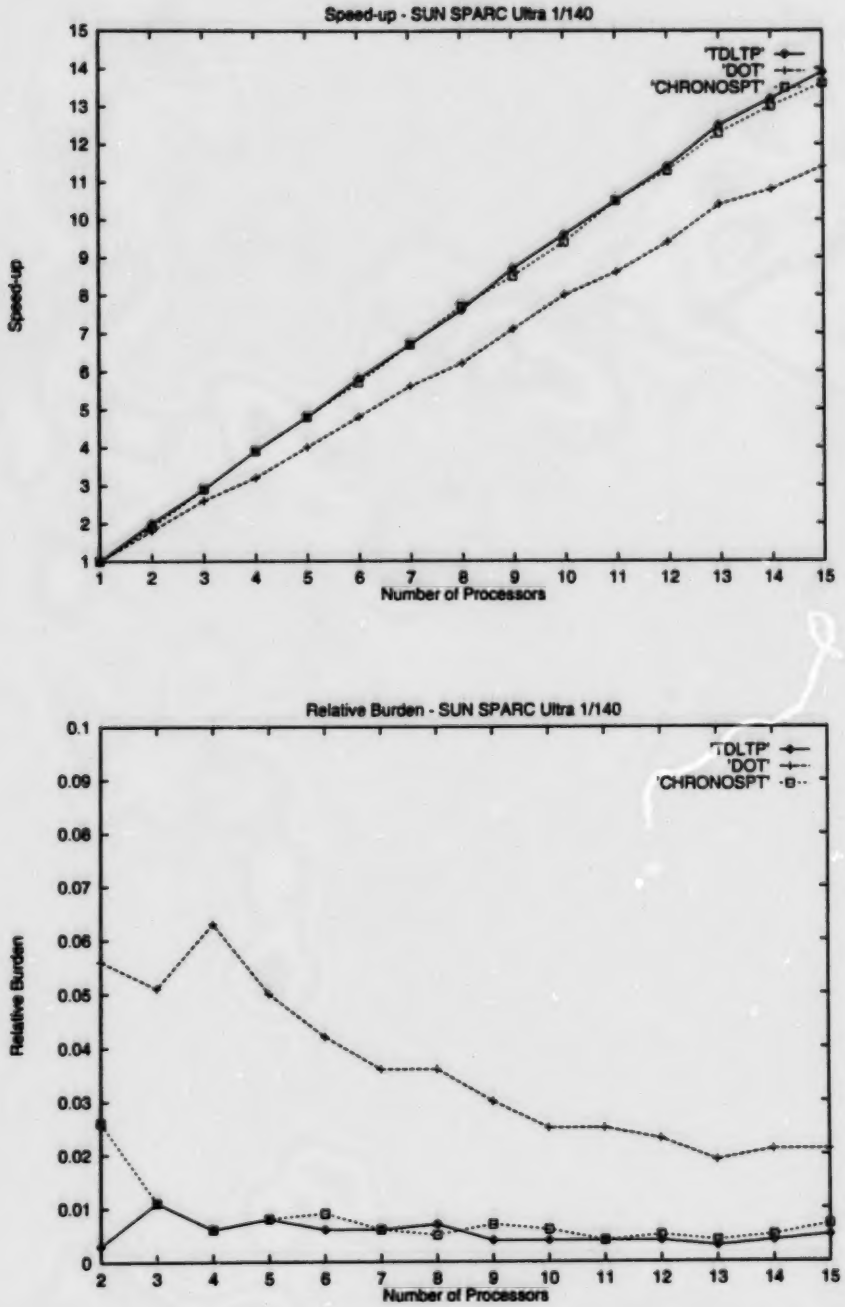
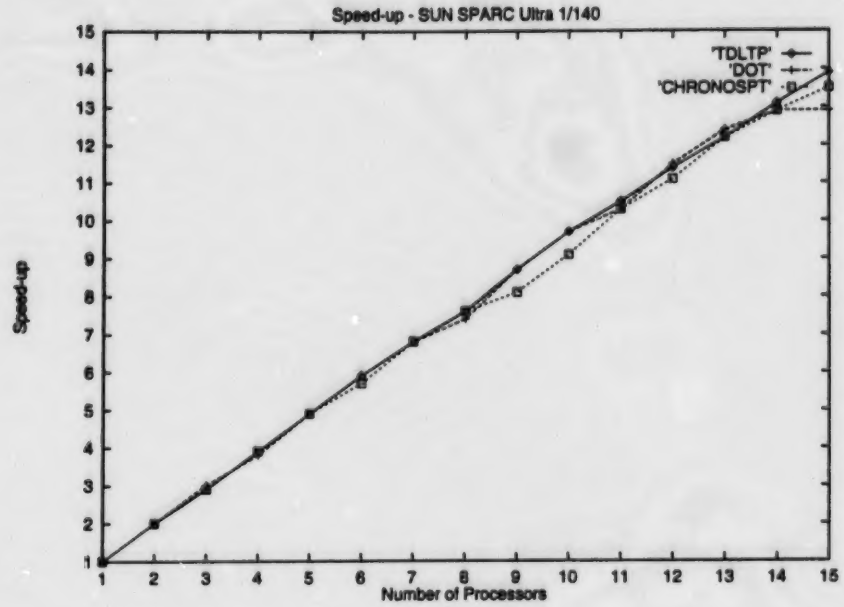


Figure 4.8 Speed-up and relative burden - Ottawa, 90 time periods



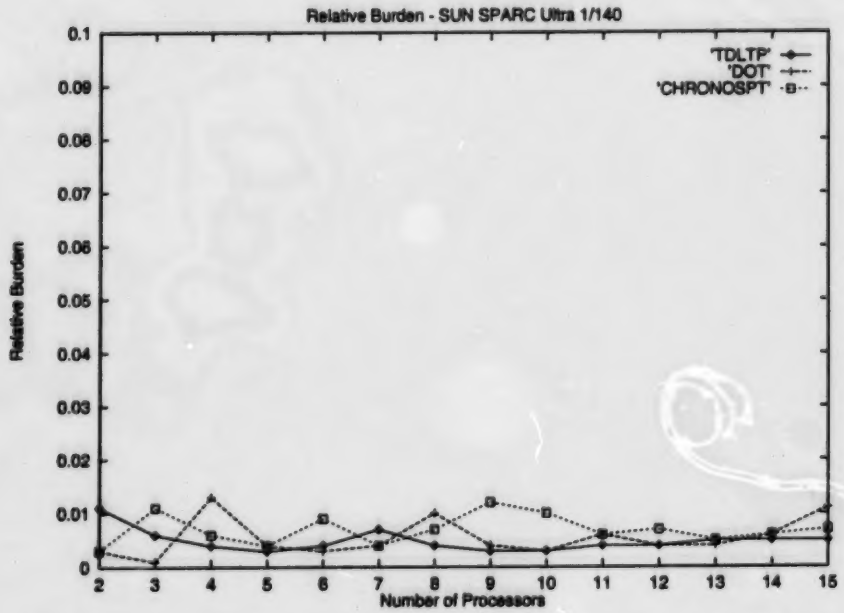
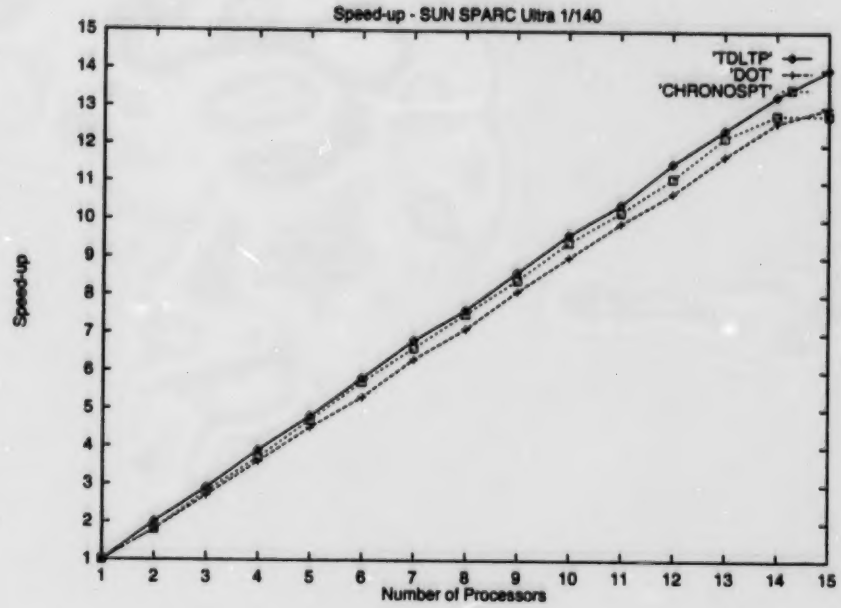


Figure 4.9 Speed-up and relative burden - Ottawa, 120 time periods



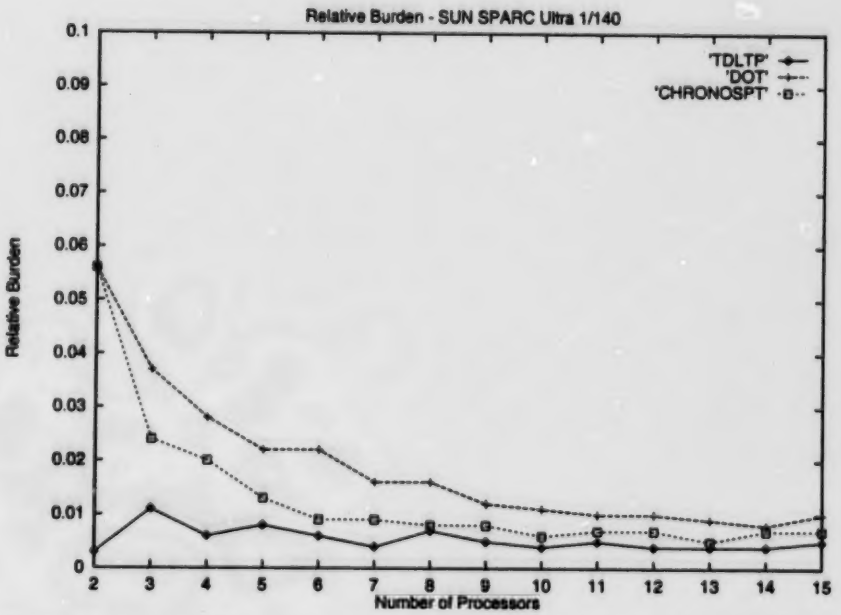
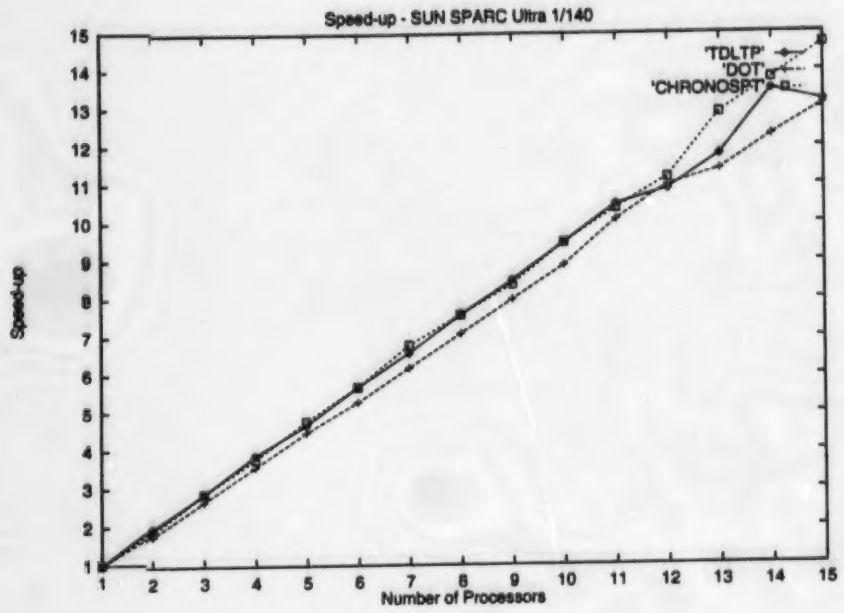


Figure 4.10 Speed-up and relative burden - Montreal, 30 time periods



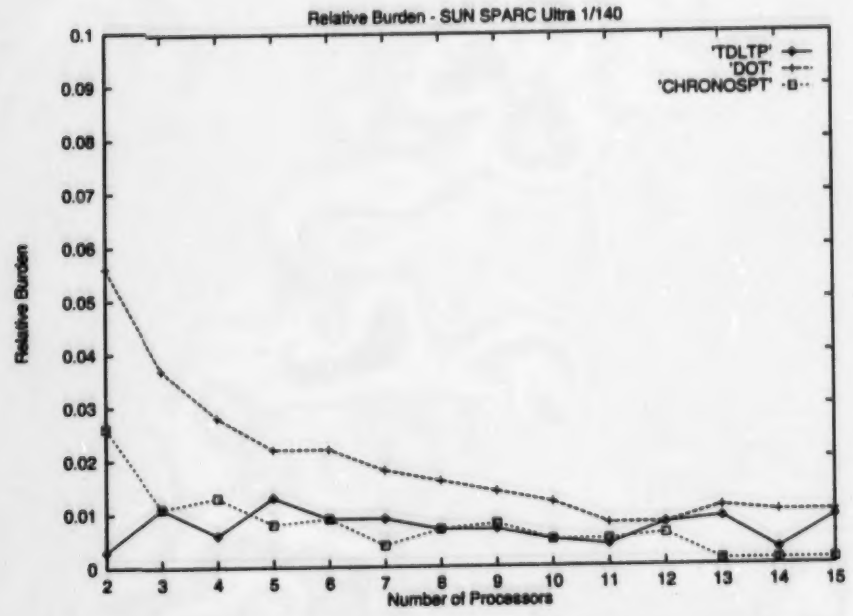


Figure 4.11 Speed-up and relative burden - Montreal, 60 time periods

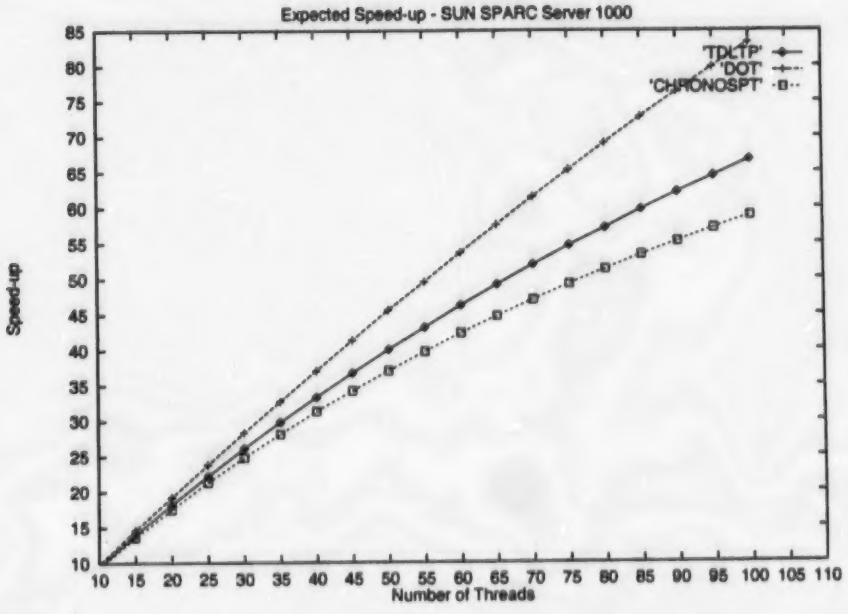
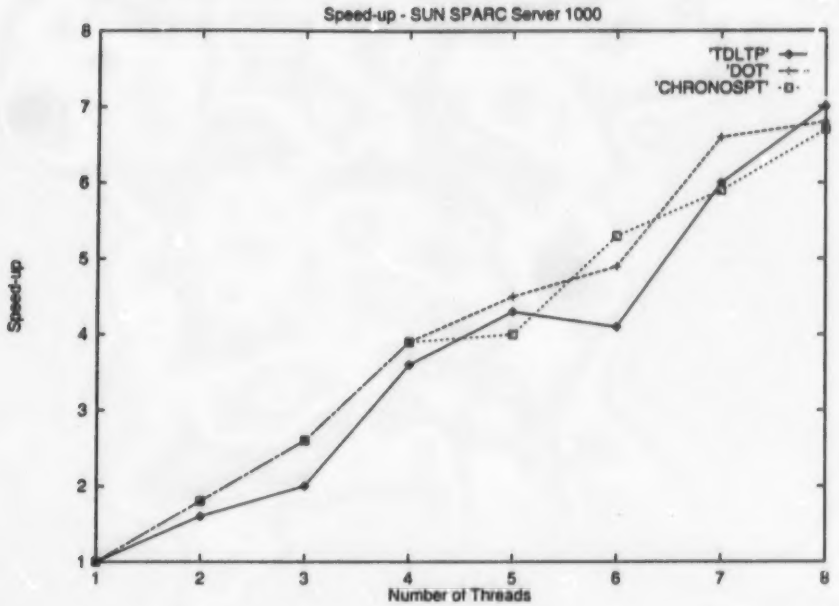


Figure 4.12 Estimated speed-up for more than 8 threads



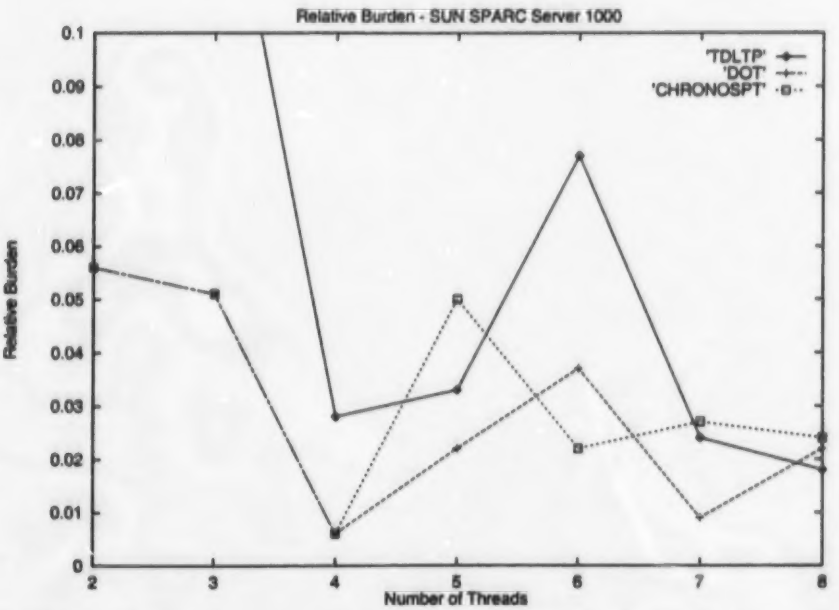
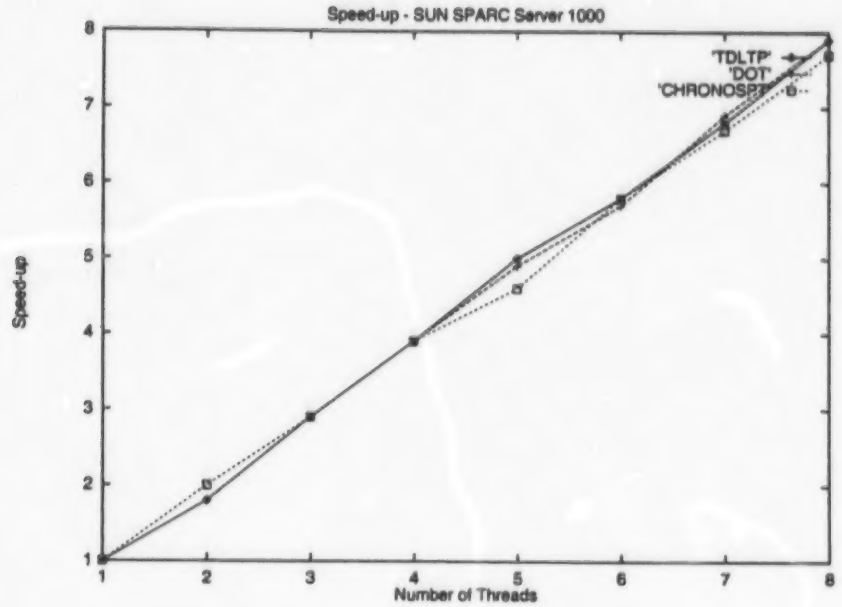


Figure 4.13 Speed-up and relative burden - Winnipeg, 30 time periods



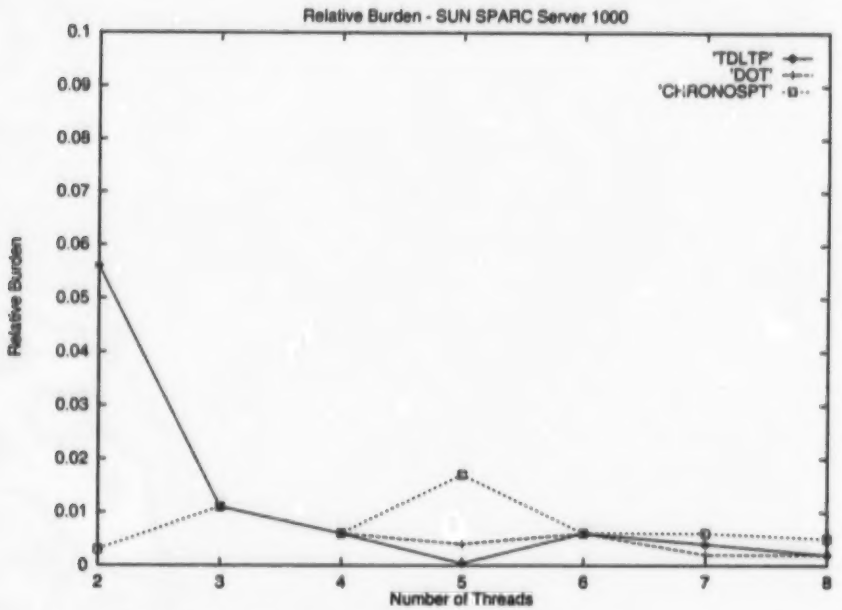
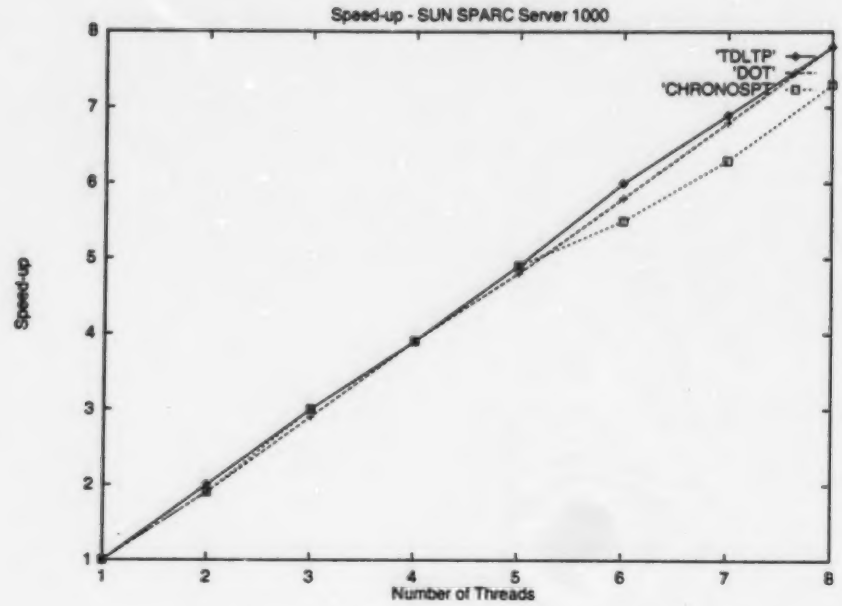


Figure 4.14 Speed-up and relative burden - Winnipeg, 60 time periods



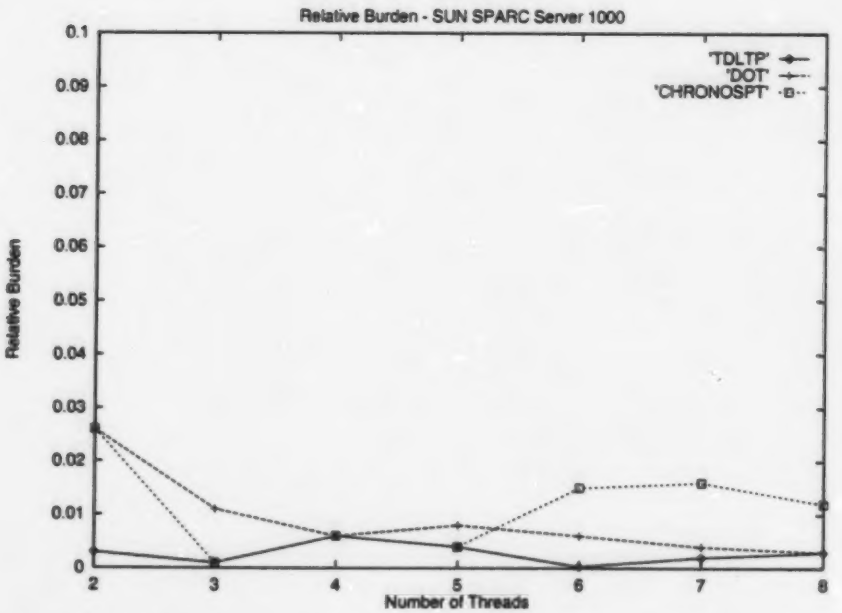
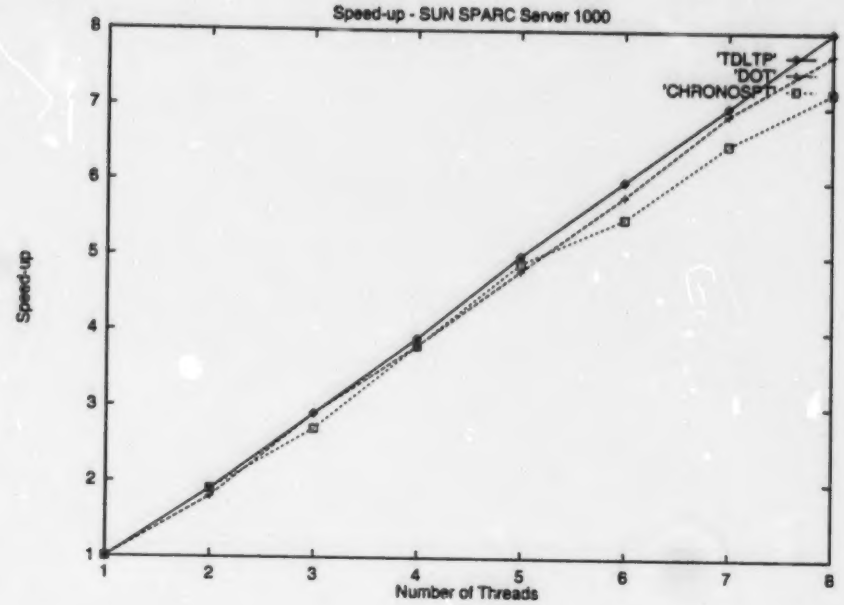


Figure 4.15 Speed-up and relative burden - Winnipeg, 90 time periods



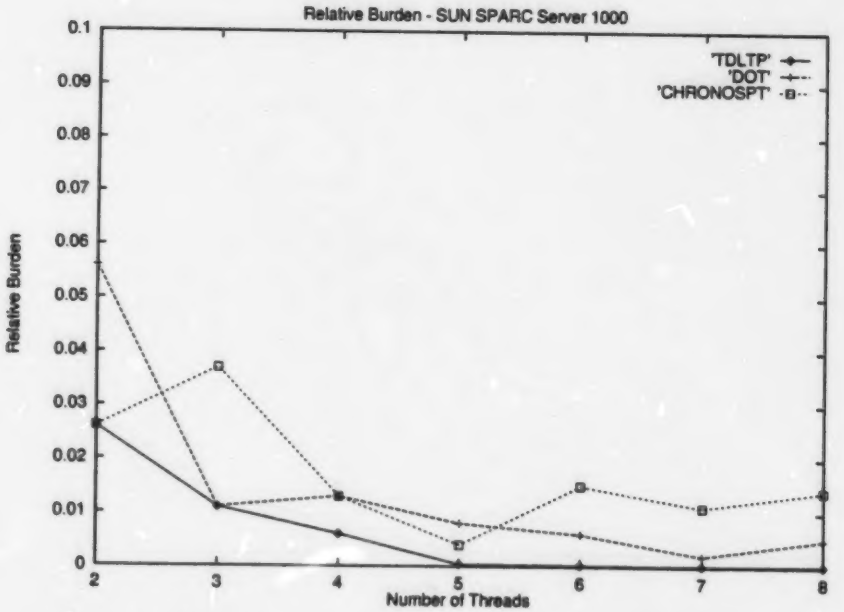
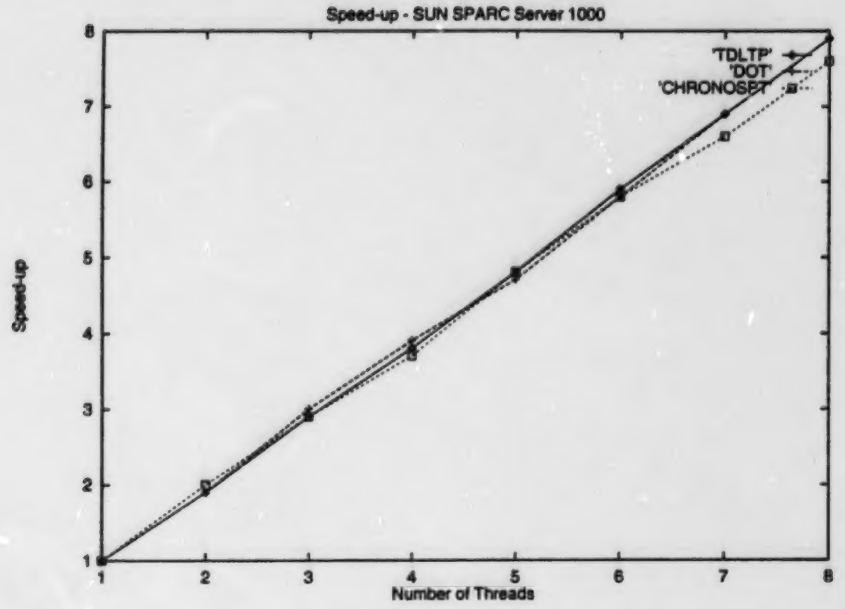


Figure 4.16 Speed-up and acceleration - Winnipeg, 120 time periods



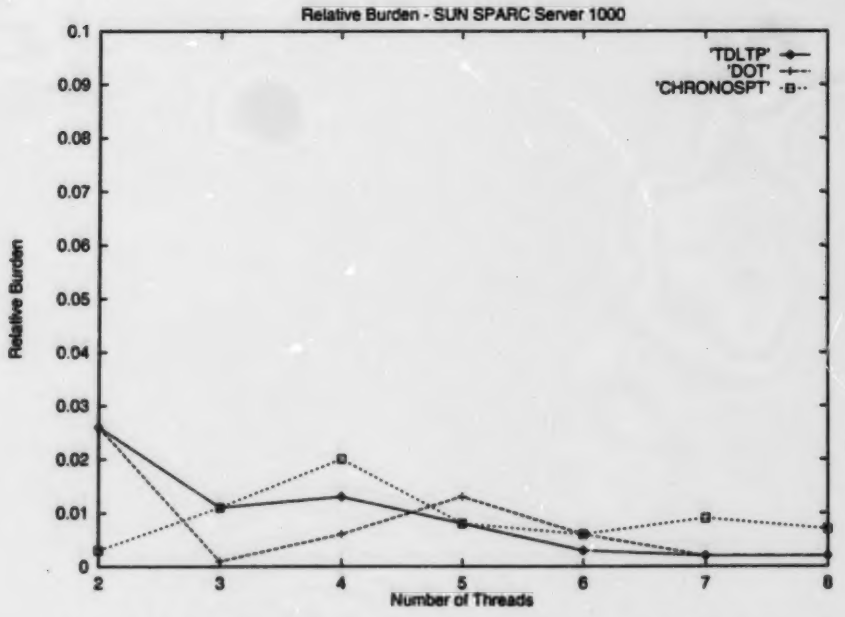
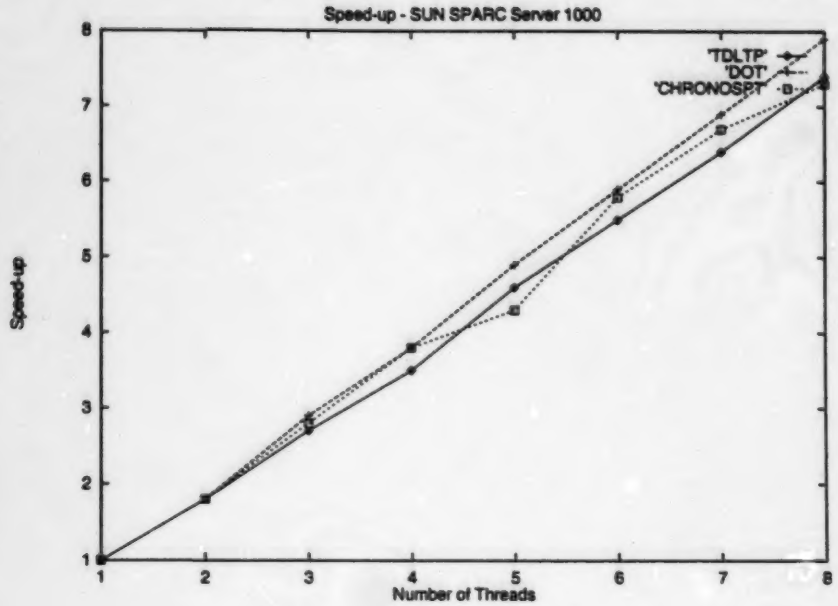


Figure 4.17 Speed-up and relative burden - Ottawa, 30 time periods



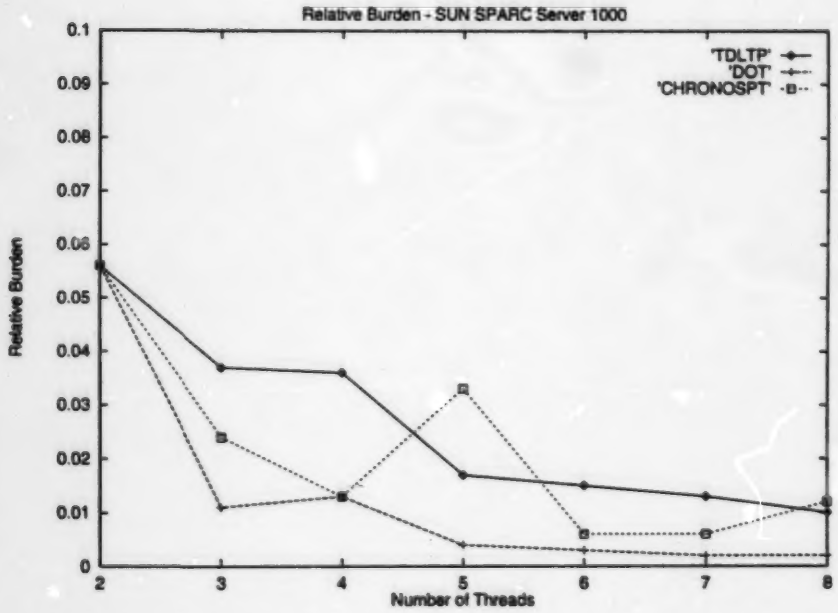
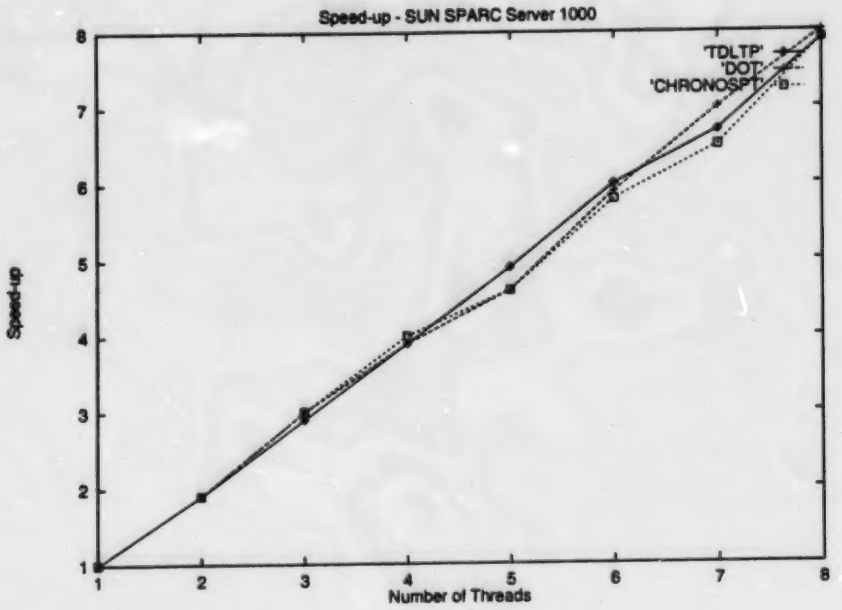


Figure 4.18 Speed-up and acceleration - Ottawa, 60 time periods



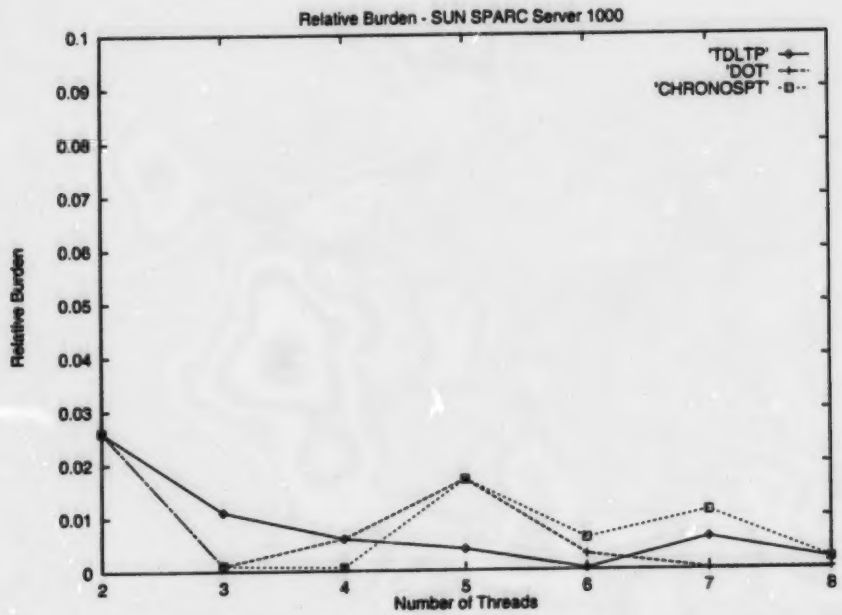
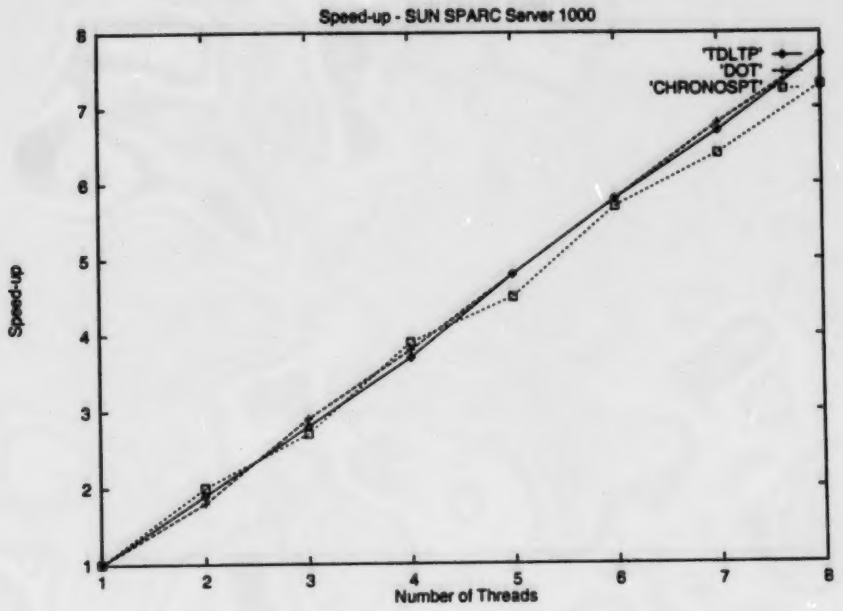


Figure 4.19 Speed-up and acceleration - Ottawa, 90 time periods



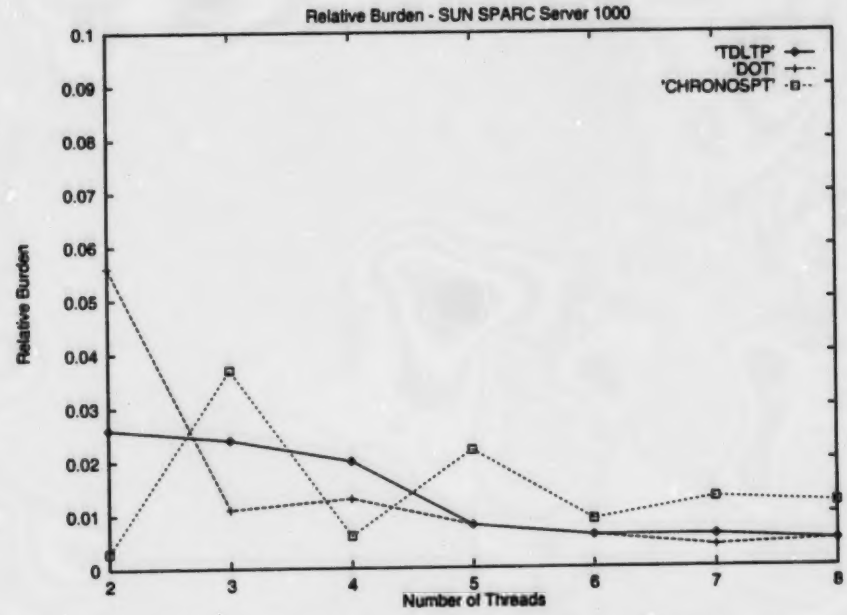
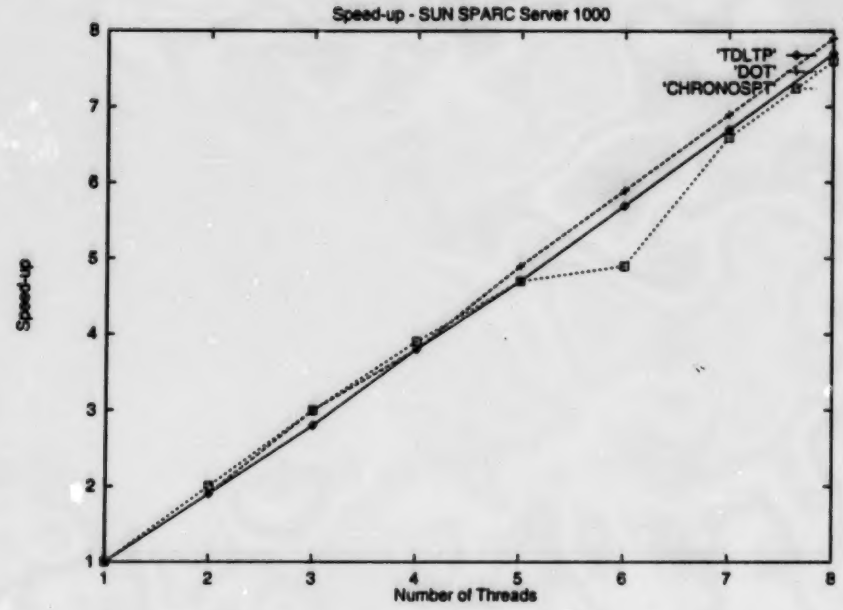


Figure 4.20 Speed-up and relative burden - Ottawa, 120 time periods



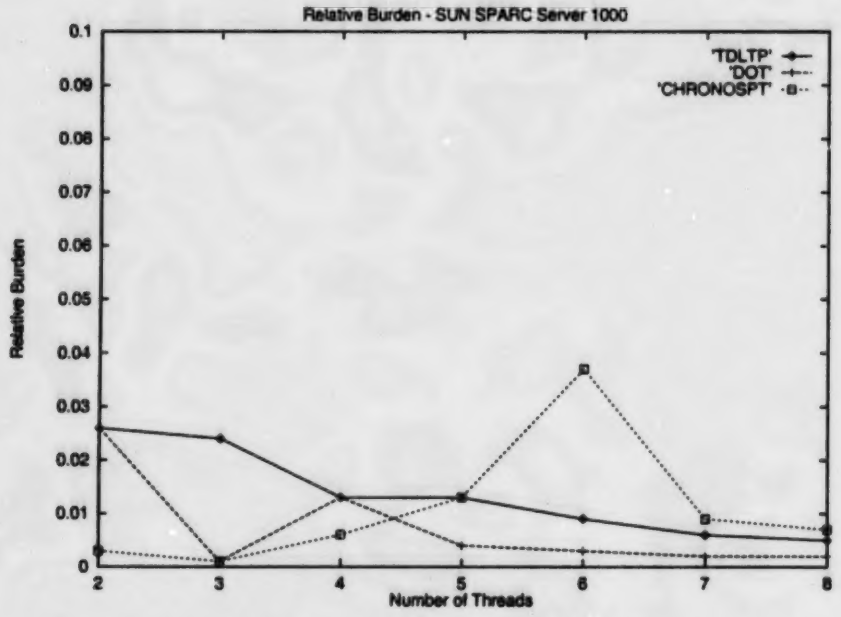
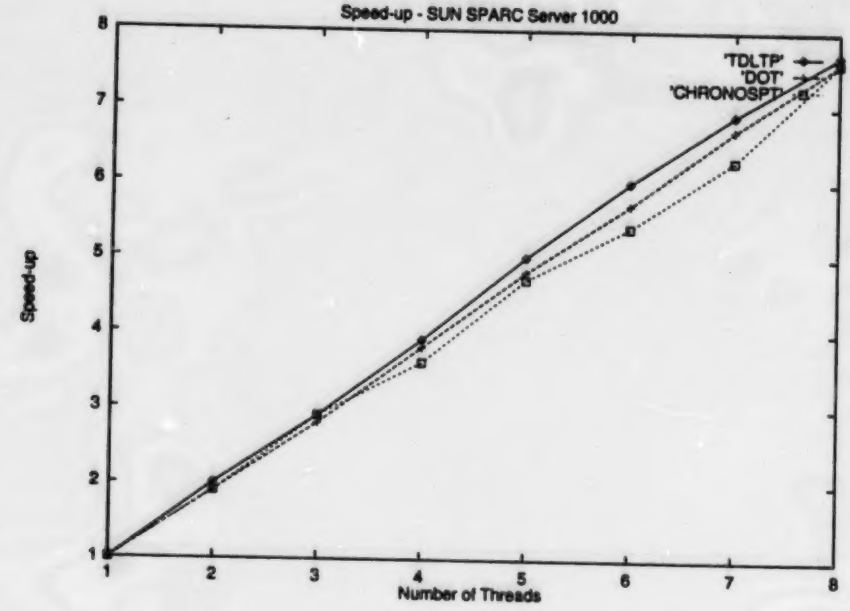


Figure 4.21 Speed-up and relative burden - Montreal, 30 time periods



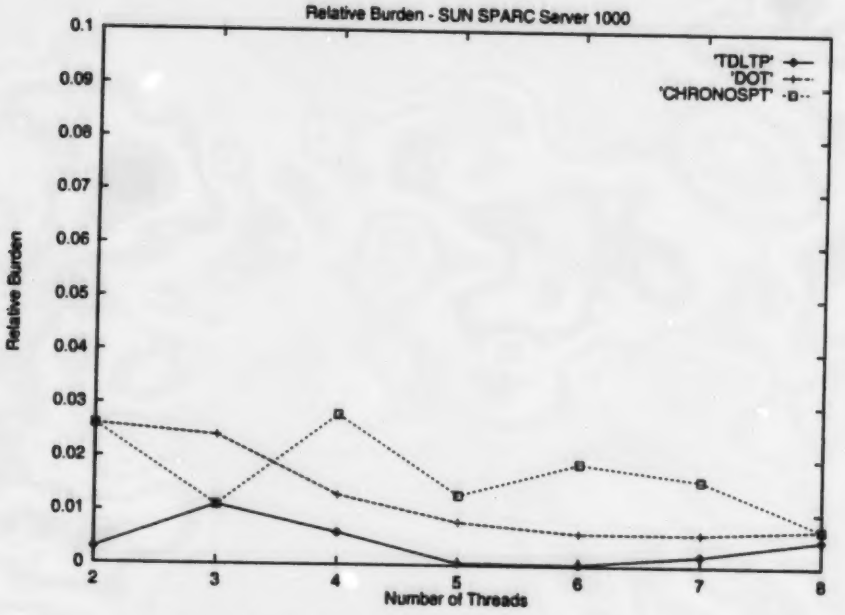
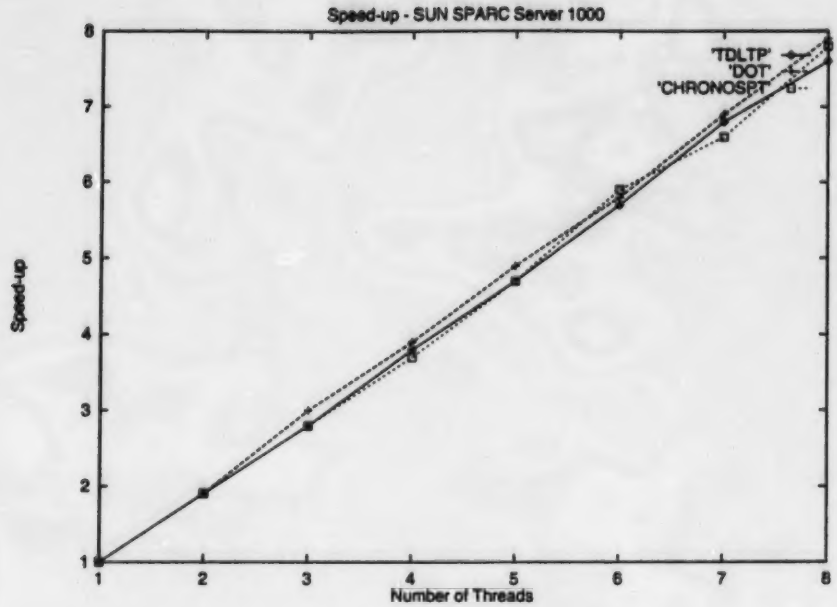


Figure 4.22 Speed-up and relative burden - Montreal, 60 time periods



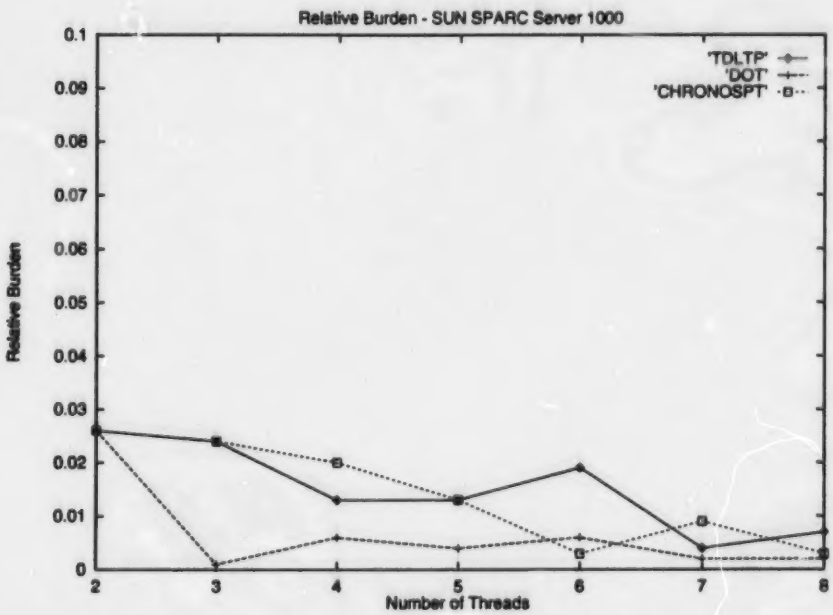
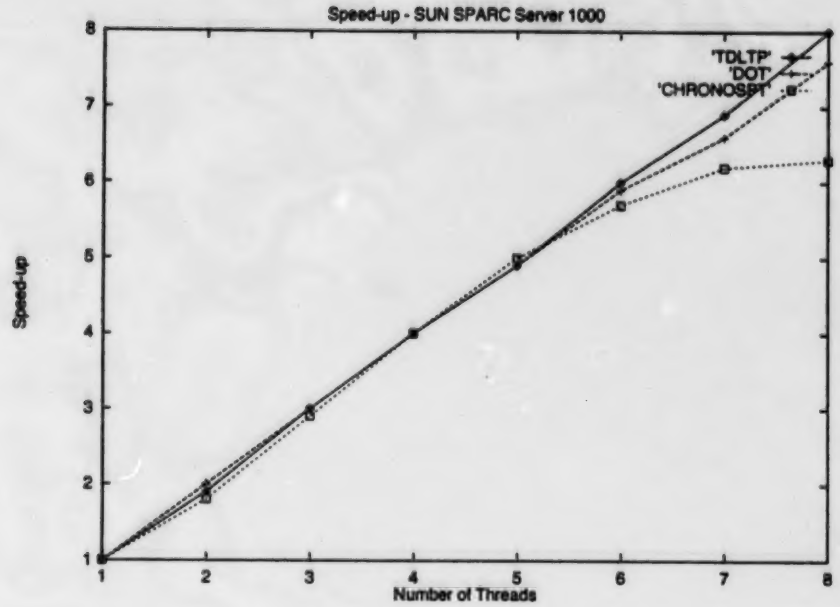


Figure 4.23 Speed-up and relative burden - Montreal, 90 time periods



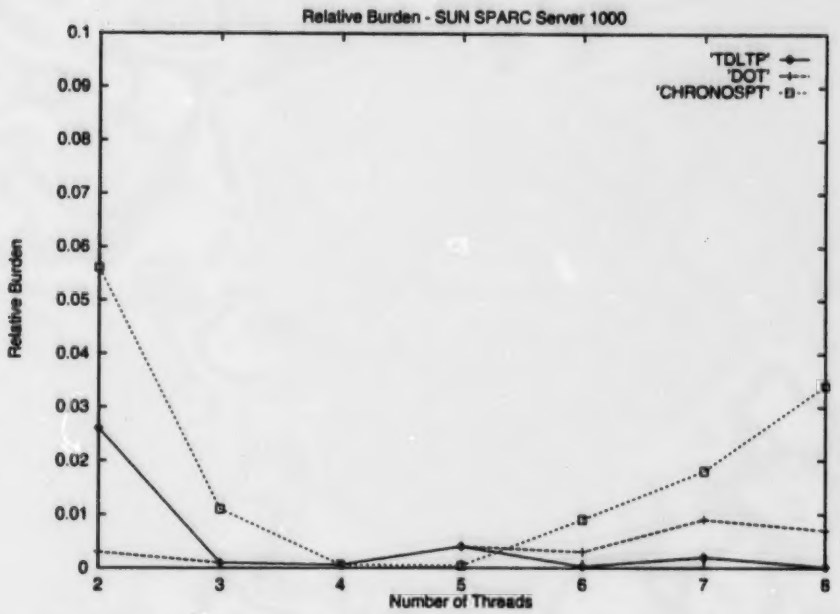


Figure 4.24 Speed-up and relative burden - Montreal, 120 time periods